SEBASTIAN FISCHER

INFORMATIK FÜR LEHRKRÄFTE

Abkömmling des Skriptes von Frank Huch zur Vorlesung 'Informatik für Nebenfächler' an der Uni Kiel angepasst und erweitert durch vertiefende Inhalte durch Sebastian Fischer im Auftrag des Instituts für Qualitätsentwicklung an Schulen Schleswig-Holstein (IQSH).

Inhaltsverzeichnis

1 *Blick über die Informatik* 5

2 *Algorithmen* 7

3 *Grundlagen der Programmierung* 11

4 *Grundlegende Programmiertechniken* 25

5 *Funktionen und Prozeduren* 37

6 *Programmierung mit Arrays* 51

7 *Rekursion* 55

8 *Syntaxbeschreibung mit (E)BNF* 63

9 *Terme und ihre Auswertung* 69

10 *Objekte und ihre Identität* 75

11 *Ruby-spezifische Sprachkonstrukte* 79

12 Definition von Objekten 83

13 Sortieren und Effizienz 91

14 Rechnerarchitektur 101

15 Digitale Bildverarbeitung 111

16 Reguläre Ausdrücke 123

17 Backtracking 127

18 Künstliche Intelligenz für Spiele 133

19 Funktionale Programmierung 147

20 Relationale Datenbanken 161

21 Netzwerke 175

22 Dynamische Webseiten 193

23 Web-Programmierung mit Ruby 201

24 Informationstheorie und Daten-Kompression 211

25 Berechenbarkeit und Komplexität 217

Index 227

1

Blick über die Informatik

Die Informatik ist die Wissenschaft von der systematischen Verarbeitung von Informationen, insbesondere der automatischen Verarbeitung mit Rechenanlagen (Computern).

Sie hat ihre Ursprünge in der **Mathematik** (Berechnen von Folgen, Lösung von Gleichungssystemen), **Elektrotechnik** (Computer als Weiterentwicklung von Schaltkreisen) und in der **Nachrichtentechnik** (Datenübertragung im Rechner oder im WWW).

Teilgebiete

Die Informatik ist aufgeteilt in die Gebiete der Theoretischen Informatik, der Praktischen Informatik, der Technischen Informatik und der Angewandten Informatik.

Die **Theoretische Informatik** ist Grundlage der anderen Teilgebiete. Sie untersucht die Entscheid- und Berechenbarkeit von Problemen, deren algorithmische Komplexität und formale Methoden zu deren Modellierung wie Automatentheorie, Graphentheorie oder Logik.

Die **Praktische Informatik** untersucht die Lösung konkreter informatischer Probleme wie die Verwaltung großer Datenmengen mit Datenstrukturen oder die Implementierung von Algorithmen mit Hilfe von Programmiersprachen.

Die **Technische Informatik** untersucht Konzepte zum Bau von Computern, also hardwareseitige Grundlagen wie Mikroprozessortechnik, Rechnerarchitektur und Netzwerksysteme.

Die **Angewandte Informatik** beschäftigt sich mit der Anwendung informatischer Methoden in Informatik-fremden und interdisziplinären Gebieten. Anders als die anderen Teilgebiete beschäftigt sie sich nicht mit Informatik-eigenen Problemstellungen.

Quellen und Lesetipps

- Informatik in der Wikipedia: `http://de.wikipedia.org/wiki/Informatik`
- Wikipedia-Artikel zu Konrad Zuse, Alan Turing, Charles Babbage, John von Neumann und Noam Chomsky

2

Algorithmen

Handlungsvorschriften zur Lösung eines Problems (oder einer Klasse von Problemen), die ausführbar, eindeutig und endlich sind, werden Algorithmen genannt. **Ausführbarkeit** verlangt, dass der Effekt jeder Anweisung eindeutig festgelegt ist. **Eindeutigkeit** verlangt, dass zu jedem Zeitpunkt der Ausführung die nächste Anweisung eindeutig festgelegt ist. **Endlichkeit** verlangt, dass die Beschreibung des Algorithmus endlich ist. Eine zusätzliche Forderung wäre **Terminierung**, d.h., dass jede Ausführung des Algorithmus nach endlich vielen Schritten endet.

Die Untersuchung konkreter Algorithmen sowie der Methoden zur Formulierung von Algorithmen in Programmiersprachen ist zentrale Aufgabe der Praktischen Informatik. Die Komplexität konkreter Algorithmen sowie abstrakte Klassen zur Einteilung von Algorithmen bezüglich ihrer Komplexität werden in der Theoretischen Informatik untersucht.

Ein interessantes Ergebnis der Theoretischen Informatik ist, dass das sogenannte **Halteproblem** nicht entscheidbar ist, d.h., dass es keinen Algorithmus gibt, der zu einem beliebigen gegebenen Algorithmus mit Eingabe entscheidet, ob dieser Algorithmus mit der Eingabe terminiert.

Algorithmen, die auf einem Computer ausgeführt werden können, heißen Computer-Programme oder kurz **Programme**.

Beispiele

- Kochrezept
- Fahrzeugsteuerung
- Rechtschreibprüfung
- Börsenkursanalyse
- Sortierverfahren (für Telefonbuch, Musik-Playlist)
- Euklidischer Algorithmus (für Division mit Rest)
- Zeichnen geometrischer Figuren

Notation

Im folgenden beschreiben wir einige Algorithmen zum Zeichnen geometrischer Figuren unter Verwendung einer beispielhaft eingeführten Notation.

Einen Algorithmus zum Zeichnen einer aus fünf Punkten im Abstand vom einem Zentimeter gezeichneten Linie notieren wir wie folgt.

```
LINIE:
  einen Zentimeter vorwärts
  Punkt zeichnen
  fünf mal wiederholen
```

Die Beschreibung zum Zeichnen einer solchen Linie ist endlich (sie hat vier Zeilen). Wenn wir vorraussetzen, dass festgelegt ist, wie die primitiven Anweisungen (z.B. `Punkt zeichnen`) ausgeführt werden, ist der Algorithmus ausführbar. Es ist auch eindeutig festgelegt, welche Anweisung auf welche andere folgt. Schließlich bemerken wir, dass der Algorithmus terminiert, denn es werden zehn Schritte (fünf mal zwei) ausgeführt.

Die folgende "Grafik" illustriert die Ausführung der Anweisung `LINIE zeichnen`.

```
  * * * * *
```

Dadurch, dass wir den Algorithmus `LINIE` genannt haben, können wir ihn als Anweisung in anderen Algorithmen verwenden. Als Beispiel definieren wir einen Algorithmus zum Zeichnen eines Quadrats.

```
QUADRAT:
  LINIE zeichnen
  rechts rum drehen
  vier mal wiederholen
```

Im Algorithmus `QUADRAT` ist die Bedeutung der Anweisung `LINIE zeichnen` ebenso vorausgesetzt, wie im Algorithmus `LINIE` die Bedeutung der Anweisung `Punkt zeichnen`. Allerdings haben wir die komplexe Anweisung `LINIE zeichnen` explizit als Algorithmus definiert, während die Bedeutung der primitiven Anweisung `Punkt zeichnen` implizit vorausgesetzt wird.

Die Benennung von Algorithmen und deren Wiederverwendung in anderen Algorithmen ist eine wichtige Form der **Abstraktion** bei der Lösung von Problemen, da es dadurch überflüssig wird, häufig wiederkehrende Anweisungsfolgen immer wieder zu notieren.

Die folgende Grafik verdeutlicht die Ausführung der Anweisung `QUADRAT zeichnen`.

```
* * * * * *
*         *
*         *
*         *
* * * * * *
```

Zur Definition des `QUADRAT`-Algorithmus haben wir implizit angenommen, dass die Anweisung `rechts rum drehen` eine Drehung um genau 90 Grad beschreibt. Wenn wir beliebige Drehungen zulassen, können wir auch Kreise zeichnen.

Der folgende Algorithmus beschreibt das Zeichnen eines Kreises mit einem Radius von fünf Zentimetern.

```
KREIS_5:
  fünf Zentimeter vor
  Punkt zeichnen
  fünf Zentimeter zurück
  ein Grad nach rechts drehen
  360 mal wiederholen
```

Veranschaulichen Sie sich die Ausführung der Anweisung `KREIS_5 zeichnen`, indem Sie entsprechend der Definition von `KREIS_5` Punkte auf ein Blatt Papier zeichnen.

Um einen Kreis mit dem Radius zehn (statt fünf) zu zeichnen, können wir in der ersten und dritten Anweisung das Wort `fünf` durch `zehn` ersetzen. Der Rest der Definition bleibt dabei unverändert. Statt für jeden konkreten Radius einen eigenen Algorithmus zu definieren, können wir aber auch einen einzigen Algorithmus zum Lösen der Problemklasse "Kreis mit gegebenem Radius zeichnen" angeben. Dazu fügen wir hinter dem Namen des Algorithmus in Klammern einen Parameter ein, der den Radius des gezeichneten Kreises festlegt.

```
KREIS(RADIUS):
  RADIUS Zentimeter vor
  Punkt zeichnen
  RADIUS Zentimeter zurück
  ein Grad nach rechts drehen
  360 mal wiederholen
```

Die Verallgemeinerung eines konkreten Problems zu einer Problemklasse durch Parametrisierung ist (wie die Wiederverwendung benannter Algorithmen als komplexe Anweisungen) eine wichtige Form der Abstraktion zur Lösung von Problemen. Sie erlaubt (potentiell unendlich!) viele gleichartige Handlungsvorschriften mit Hilfe eines einzigen Algorithmus zu beschreiben.

Quellen und Lesetipps

- Wikipedia-Artikel: Algorithmus, Muhammed al-Chwarizmi, Ada Lovelace
- Lauren Ipsum, A story about computer science and other improbable things

3

Grundlagen der Programmierung

Algorithmen heißen Programme, wenn sie automatisch von einem Computer ausgeführt werden können. Dazu müssen sie in einer Programmiersprache implementiert werden.

Im folgenden werden wir uns mit sogenannten höheren Programmierkonzepten am Beispiel der Programmiersprache Ruby beschäftigen. Solche Sprachen stellen spezielle Konstrukte bereit, die die bedingte oder wiederholte Ausführung von Anweisungs-Sequenzen ermöglichen. Neben diesen sogenannten Kontrollstrukturen bieten höhere Programmiersprachen die Möglichkeit, komplexe (zum Beispiel arithmetische) Ausdrücke zu notieren und automatisch auszuwerten.

Arithmetische Ausdrücke und Variablen

In Ruby können Zahlen als primitive Werte verwendet werden. Sie werden dabei automatisch in einer geeigneten Darstellung im Speicher abgelegt. Wie genau die Daten intern dargestellt werden, ist bei der Programmierung in der Regel irrelevant. Es genügt, vordefinierte Funktionen und Operationen zu kennen, mit denen wir Zahlen verarbeiten können.

Durch Verknüpfung mit Funktionen und Operationen entstehen komplexe Ausdrücke, die von Ruby automatisch ausgewertet werden. Die interaktive Ruby-Umgebung “irb” erlaubt es, beliebigen Ruby-Code in einem Terminal auszuführen, kann also auch dazu verwendet werden, arithmetische Ausdrücke auszuwerten.

```
irb> 3 + 4
7
```

Arithmetische Ausdrücke

Aus der Mathematik kennen wir Ausdrücke wie $x^2 + 2y + 1$ oder $(x + 1)^2$, die auch Variablen enthalten können. Diese entstehen

aus Basiselementen

- Konstanten (z.B. $1, 2$ oder π)
- Variablen (z.B. x, y)

und können durch Anwendung von Funktionen wie $+, -, \cdot$ auf bereits existierende Ausdrücke gebildet werden. Diese Funktionen (auch Operatoren genannt) sind zweistellig, verknüpfen also zwei Ausdrücke zu einem neuen Ausdruck.

Auch der Ausdruck $\frac{\sqrt{x^2+1}}{x}$ entsteht durch Anwendung unterschiedlicher Funktionen, allerdings ungewöhnlich notiert. Ruby erfordert eine einheitlichere Darstellung von Ausdrücken. Zum Beispiel müssen wir

- `x**2` statt x^2,
- `Math.sqrt(x)` statt $\sqrt{x}$ und
- `a/b` statt $\frac{a}{b}$

schreiben. Den Ausdruck $\frac{\sqrt{x^2+1}}{x}$ schreiben wir in Ruby also als `Math.sqrt(x**2+1)/x`. Hierbei können wir durch festgelegte Präzedenzen (Punktrechnung vor Strichrechnung) auf Klammern verzichten. Schreiben wir stattdessen `Math.sqrt(x**(2+1))/x`, so ergibt sich *nicht* der gleiche Ausdruck, da die Funktion `**` stärker bindet als `+`.

Im folgenden werten wir beispielhaft einige arithmetische Ausdrücke in der “irb”-Umgebung aus.

```
irb> 3**2
=> 9
irb> Math.sqrt(25)
=> 5.0
irb> 9/3
=> 3
irb> Math.sqrt(5**2-9)/4
=> 1.0
```

Variablen und Zuweisungen

In der Mathematik können arithmetische Ausdrücke Variablen enthalten, die als Platzhalter für unbekannte Werte stehen. Auch in Programmiersprachen können wir Variablen verwenden, wenn wir ihnen vorher einen Wert zuweisen.

Als Beispiel für einen Ausdruck mit Variablen betrachten wir die Formel $\pi \cdot r^2$ zur Bestimmung des Flächeninhalts eines Kreises mit gegebenem Radius r.

In Ruby können wir diese Formel wie folgt schreiben.

```
irb> Math::PI * r**2
NameError: undefined local variable or method 'r' for main:Object
```

Da wir der Variablen `r` jedoch noch keinen Wert zugewiesen haben, liefert “irb” beim Versuch, die Formel auszuwerten, eine Fehlermeldung. Durch Zuweisung verschiedener Werte an `r` können wir den Flächeninhalt von Kreisen mit unterschiedlichen Radien berechnen.

```
irb> r = 2
irb> Math::PI * r**2
=> 12.5663706143592
irb> r = 4
irb> Math::PI * r**2
=> 50.2654824574367
```

Die Zeilen `r = 2` und `r = 4` sind anders als alles bisher eingegebene keine Ausdrücke sondern Zuweisungen, also eine spezielle Form sogenannter Anweisungen oder Instruktionen. Anweisungen haben anders als Ausdrücke keinen Wert[1]. Zuweisungen speichern den Wert des Ausdrucks rechts vom Gleichheitszeichen in der Variablen links vom Gleichheitszeichen.

[1] Tatsächlich haben in Ruby Anweisungen genau wie Ausdrücke Werte. Wir unterscheiden trotzdem konzeptuell zwischen Ausdrücken, die einen Wert haben, und Anweisungen, die keinen haben.

Während in der Mathematik die Gleichung $x = x + 1$ keine Lösungen hat, ist die Zuweisung `x = x + 1` durchaus üblich:

```
irb> x = 41
irb> x
=> 41
irb> x = x + 1
irb> x
=> 42
```

Sie weist der Variablen `x` den Wert `x + 1` zu, also ihren eigenen um eins erhöhten (alten) Wert.

Weitere primitive Datentypen

In Ruby können wir nicht nur arithmetische sondern zum Beispiel auch logische Ausdrücke auswerten und solche, deren Wert eine Zeichenkette, also Text, ist.

Zeichenketten

Eine Zeichenkette (englisch: string) wird dazu in Anführungszeichen eingeschlossen. Mehrere Zeichenketten können mit dem +-Operator aneinandergehängt werden.

```
irb> "Hallo"
=> "Hallo"
irb> "Welt"
=> "Welt"
irb> "Hallo" + "Welt"
=> "HalloWelt"
irb> "Hallo" + " " + "Welt"
=> "Hallo Welt"
irb> "Hallo" + " " + "Welt" + "!"
=> "Hallo Welt!"
```

Zahlen können wir in Zeichenketten konvertieren, indem wir ihnen `.to_s` (für **to** __s__tring) anhängen. Auf diese Weise können wir Zeichenketten mit arithmetischen Ausdrücken kombinieren.

```
irb> 42.to_s
=> "42"
irb> (17+4).to_s
=> "21"
irb> 17.to_s + 4.to_s
=> "174"
irb> "Die Antwort ist " + (2*(17+4)).to_s
=> "Die Antwort ist 42"
```

Der Operator + wird also sowohl zur Addition von Zahlen als auch zur Konkatenation von Zeichenketten verwendet. Beim Versuch + mit einer Zahl und einer Zeichenkette aufzurufen, erhalten wir allerdings einen Fehler.

```
irb> "40" + 2
TypeError: can't convert Fixnum into String
irb> 40 + "2"
TypeError: String can't be coerced into Fixnum
```

Die Fehlermeldungen deuten darauf hin, dass Zahlen und Zeichenketten nicht automatisch ineinander konvertiert werden, denn es ist unklar, ob als Ergebnis die Zahl `42` oder die Zeichenkette `"402"` herauskommen soll. Diese Unklarheit müssen wir durch explizite Konvertierung (mittels `to_s`) aufklären. Wollen wir eine Zeichenkette, die eine Zahl enthält, in eine Zahl konvertieren, können wir `.to_i` (für **to** __i__nteger) oder `.to_f` (für **to** __f__loating point number) benutzen.

```
irb> "40".to_i + 2
=> 42
irb> "40" + 2.to_s
=> "402"
irb> 40 + "2".to_f
=> 42.0
```

Logische Ausdrücke

Logische Ausdrücke beschreiben Wahrheitswerte. Sie sind aus den Konstanten `true` und `false` aufgebaut, wobei komplexere Ausdrücke durch Anwendung logischer Operationen gebildet werden können.

Eine Konjunktion (logisches "und") wird durch den Operator `&&` gebildet, eine Disjunktion (logisches "oder") durch `||` und eine Negation (logisches "nicht") durch ein vorangestelltes Ausrufezeichen.

Hier sind einige Beispiele für logische Ausdrücke in der "irb"-Umgebung.

```
irb> true && false
=> false
irb> !false
=> true
irb> false || (true && !false)
=> true
```

Auch Vergleichsoperatoren haben logische Werte als Ergebnis. Zum Beispiel liefert der Ausdruck `3 < 4` das Ergebnis `true`. Hier sind weitere Beispiele für logische Ausdrücke mit Vergleichsoperatoren.

```
irb> 4 < 3
=> false
irb> 5+2 >= 6
=> true
irb> 5+2 >= 6+3
=> false
irb> 5+2 >= 6+3 || 3 <= 10/2
=> true
irb> 5+2 >= 6+3 || 3 <= 10/5
=> false
```

Variablen für Text und Wahrheitswerte

Auf rechten Seiten einer Zuweisung können beliebig komplizierte Ausdrücke stehen, deren Werte nicht unbedingt Zahlen zu sein brauchen.

```
irb> antwort = 2*(17+4)
irb> antwort
=> 42
irb> text = "Die Antwort ist " + antwort.to_s
irb> text
```

```
=> "Die Antwort ist 42"
irb> antwort = text == "Die Antwort ist 42"
irb> antwort
=> true
```

Hier wird der Variablen `antwort` zunächst der Wert `42` zugewiesen und dieser dann zur Definition der Variablen `text` verwendet. Schließlich wird der Wert der Variablen `antwort` auf `true` geändert, indem ihr das Ergebnis eines Vergleiches zugewiesen wird.

Bedingte Anweisungen

Nachdem wir im vorherigen Abschnitt Zuweisungen kennen gelernt haben, mit denen der Wert eines Ausdrucks in einer Variablen gespeichert werden kann, wenden wir uns nun einer weiteren Form der Anweisung zu. In bedingten Anweisungen ist die Ausführung einzelner Anweisungen vom Wert eines logischen Ausdrucks abhängig.

Die folgende Anweisung, in der `if`, `then` und `end` Schlüsselworte[2] sind, demonstriert diese Idee.

[2] Schlüsselworte sind von einer Programmiersprache reservierte Namen mit besonderer Bedeutung. Sie dürfen deshalb nicht als Variablennamen verwendet werden.

```
if x < 0 then x = -1 * x end
```

Hier wird die Anweisung `x = -1 * x` nur dann ausgeführt, wenn der Wert des logischen Ausdrucks `x < 0` gleich `true` ist, wenn also der Wert von `x` kleiner als Null ist. Ist das nicht der Fall (ist also der Wert des logischen Ausdrucks `x < 0` gleich `false`) dann wird die Zuweisung `x = -1 * x` nicht ausgeführt. In jedem Fall hat also nach der Ausführung der bedingten Anweisung die Variable `x` einen nicht-negativen Wert, nämlich den Absolutbetrag ihres ursprünglichen Wertes. Die folgenden Aufrufe demonstrieren die Auswertung dieser bedingten Anweisung.

```
irb> x = -4
irb> x
=> -4
irb> if x < 0 then x = -1 * x end
irb> x
=> 4
irb> if x < 0 then x = -1 * x end
irb> x
=> 4
```

Bedingte Anweisungen können auch Alternativen enthalten, die ausgeführt werden, wenn die Bedingung *nicht* erfüllt ist. Dazu verwenden wir das Schlüsselwort `else` wie im folgenden Beispiel.

```
if x > y then z = x else z = y end
```

Hier wird der Variablen `z` der Wert der Variablen `x` zugewiesen, falls dieser größer ist als der Wert von `y`. Ist das nicht der Fall, erhält `z` den Wert von `y`. In jedem Fall hat also die Variable `z` nach dieser Anweisung den Wert des Maximums der Werte von `x` und `y`.

Die folgende Anweisungsfolge demonstriert die Auswertung einer solchen Berechnung.

```
irb> x = 4
irb> y = 5
irb> if x > y then z = x else z = y end
irb> z
=> 5
```

Da der Wert des logischen Ausdrucks `x > y` gleich `false` ist, wird die Zuweisung `z = y` ausgeführt. Danach hat die Variable `z` also den Wert `5`.

Bedingte Anweisungen mit Alternative werden *Bedingte Verzweigungen* genannt. Bedingte Anweisungen ohne Alternative heißen auch *Optionale Anweisungen*.

Statt Anweisungen in der interaktiven Ruby-Umgebung “irb” einzugeben, können wir sie auch in einer Textdatei speichern. Dabei können wir einzelne Anweisungen auf mehrere Zeilen verteilen, was der Lesbarkeit des Programms zugute kommt.

Wir können zum Beispiel das folgende Programm in einer Datei `max.rb` speichern.

```
x = 4
y = 5
if x > y then
  z = x
else
  z = y
end
puts(z)
```

Hier ist die bedingte Anweisung auf mehrere Zeilen verteilt und die Alternativen sind eingerückt um sie als Teile der bedingten Anweisung hervorzuheben.

Die Ausgabe-Anweisung `puts(z)` dient dazu, den Wert von `z` im Terminal auszugeben (`puts` steht für **put** __s__tring). Dies ist nötig, da wir das Programm nicht in einer interaktiven Umgebung, die Ergebnisse von Ausdrücken automatisch anzeigt, sondern mit dem Interpreter `ruby` auswerten. Dazu wechseln

wir im Terminal in das Verzeichnis, in dem wir das Programm max.rb gespeichert haben und führen es dann mit dem folgenden Kommando aus.

```
# ruby max.rb
5
```

Als letzter Schritt der Ausführung des Programms wird die Zahl 5 im Terminal ausgegeben.

Bedingte Anweisungen eingerückt zu notieren zahlt sich besonders dann aus, wenn diese geschachtelt sind - wenn also die Alternativen selbst auch wieder bedingte Anweisungen sind. Als Beispiel einer geschachtelten bedingten Anweisung betrachten wir das folgende Programm xor.rb, das das Ergebnis der Exklusiv-Oder-Verknüpfung zweier Variablen ausgibt.

```
x = true
y = false
if x then
  if y then
    z = false
  else
    z = true
  end
else
  z = y
end
puts(x.to_s + " xor " + y.to_s + " = " + z.to_s)
```

Hier werden die logischen Ausdrücke x und y als Bedingungen für bedingte Anweisungen verwendet, wobei die zweite im sogenannten then-Zweig der ersten steht. Bei der Ausführung dieses Programms wird das Ergebnis von true xor false ausgegeben.

```
# ruby xor.rb
true xor false = true
```

Schleifen

Neben bedingten Anweisungen gibt es in höheren Programmiersprachen auch Sprach-Konstrukte zur wiederholten Ausführung von Anweisungen. Im folgenden werden zwei verschiedene solcher Konstrukte vorgestellt: die Zähl-Schleife und die bedingte Schleife.

Zähl-Schleifen

Eine Zähl-Schleife wiederholt eine Anweisung (oder einen Anweisungsblock), wobei eine Zählvariable einen festgelegten Zahlenbereich durchläuft. Die Anzahl der Wiederholungen ist also durch den definierten Zahlenbereich festgelegt.

Als Beispiel für eine Zähl-Schleife schreiben wir ein Programm `1bis100.rb`, das die Zahlen von 1 bis 100 addiert.

```
sum = 0
for i in 1..100 do
  sum = sum + i
end
puts(sum)
```

Hier sind `for`, `in`, `do` und `end` Schlüsselworte. Die sogenannte Zählvariable `i` nimmt während der wiederholten Ausführung des sogenannten Schleifenrumpfes `sum = sum + i` nacheinandner die Werte von `1` bis `100` an, so dass in `sum` nach Ausführung der Schleife die Summe der Zahlen von 1 bis 100 gespeichert ist, die mit der letzten Anweisung ausgegeben wird.

```
# ruby 1bis100.rb
5050
```

Als Grenzen für den von der Zählvariable durchlaufenen Zahlenbereich können wir beliebige Ausdrücke verwenden, deren Wert eine Zahl ist - insbesondere auch Variablen, wie das folgende Beispiel zeigt.

```
n = 7
q = 0
for i in 1..n do
  u = 2*i - 1
  q = q + u
end
puts(n.to_s + " zum Quadrat ist " + q.to_s)
```

Bei diesem Programm besteht der Schleifenrumpf aus zwei Zuweisungen. Die erste definiert `u` als die `i`-te ungerade Zahl und die zweite addiert diese zur Variablen `q` hinzu. Nach Ausführung der Schleife ist in `q` also die Summe der ersten `n` ungeraden Zahlen gespeichert, also `n` zum Quadrat.

Wenn wir dieses Programm in der Datei `quadrat.rb` speichern und diese dann ausführen, erhalten wir die folgende Ausgabe.

```
# ruby quadrat.rb
7 zum Quadrat ist 49
```

Bedingte Schleifen

Sogenannte bedingte Schleifen sind ein weiteres Konstrukt höherer Programmiersprachen zur Wiederholung von Anweisungen. Anders als bei Zähl-Schleifen hängt die Anzahl der Wiederholungen bei einer bedingten Schleife nicht von einem vorab definierten Zahlenbereich ab, sondern von einem logischen Ausdruck, der vor jedem Schleifendurchlauf ausgewertet wird. Ist der Wert dieser sogenannten Schleifenbedingung gleich `true`, so wird der Rumpf (ein weiteres Mal) ausgeführt, ist er gleich `false`, so wird die Ausführung der bedingten Schleife beendet. Bei einer bedingten Schleife ist also nicht immer vorab klar, wie oft der Schleifenrumpf ausgeführt wird, da der Wert der Bedingung von Zuweisungen im Schleifenrumpf abhängen kann.

Als erstes Beispiel für eine bedingte Schleife berechnen wir wieder die Summe der Zahlen von 1 bis 100.

```
i = 0
sum = 0
while i < 100 do
  i = i + 1
  sum = sum + i
end
puts(sum)
```

Zwischen den Schlüsselwörtern `while` und `do` steht die Schleifenbedingung, danach folgt bis zum `end` der Schleifenrumpf. Anders als mit der Zähl-Schleife müssen wir hier den Wert der Zählvariable `i` explizit setzen, da bedingte Schleifen keine eingebaute Zählvariable haben. Wenn `i` gleich `100` ist, wird die Schleife beendet und die Summe der ersten 100 Zahlen ausgegeben.

In diesem Beispiel ist die Anzahl der Schleifendurchläufe einfach ersichtlich, da die Schleifenbedingung nur von dem Wert der Variablen `i` abhangt, die in jedem Schleifendurchlauf um eins erhöht wird. Im folgenden Programm ist die Anzahl der Schleifendurchläufe nicht so einfach ersichtlich.

```
n = 144
i = 0
q = 0
while q < n do
  i = i + 1
  q = q + 2*i - 1
end
puts(i)
```

Hier wird in jedem Durchlauf die Zählvariable `i` um eins erhöht und (wie beim Programm `quadrat.rb`) der Variablen `q` die `i`-te ungerade Zahl hinzuaddiert. Die Schleife wird ausgeführt, solange der Wert von `q` kleiner als `n` ist. Sie bricht also ab, sobald `q` größer oder gleich `n` ist.

Wie im Programm `quadrat.rb` ist nach jedem Schleifendurchlauf `q = i*i`. Das obige Programm gibt also die kleinste Zahl `i` aus, deren Quadrat größer oder gleich `n` ist. Ist `n` eine Quadratzahl, so ist die Ausgabe des Programms deren Quadratwurzel.

```
# ruby wurzel.rb
12
```

Bei der Programmierung mit bedingten Schleifen ist Vorsicht geboten, da nicht sichergestellt ist, dass die Schleifenbedingung irgendwann nicht mehr erfüllt ist. In diesem Fall bricht die Schleife nie ab, läuft also (potentiell) endlos weiter.

Eine einfache Endslosschleife können wir wie folgt definieren.

```
while true do
end
```

Da diese Schleife nie beendet wird, werden nach ihr folgende Anweisungen nie ausgeführt. Eine häufige Fehlerquelle sind Zählvariablen, die wir vergessen im Rumpf zu erhöhen. Auch das folgende Programm terminiert also nicht.

```
i = 0
sum = 0
while i < 100 do
  sum = sum + i
end
```

Um versehentliche Nicht-Terminierung von vornherein auszuschließen sollten Sie wenn möglich Zähl-Schleifen verwenden. Nur wenn die Anzahl der Schleifendurchläufe nicht (einfach) ersichtlich ist, sollten Sie auf bedingte Schleifen zurückgreifen.

Tabellarische Programmausführung

In diesem Abschnitt lernen wir eine systematische Methode kennen, die Ausführung eines Programms zu dokumentieren. Sich im Kopf zu überlegen, welche Variablen wann mit welchen Werten belegt sind, wird bei größeren Programmen schnell unübersichtlich. Übersichtlicher ist eine tabellarische Notation, die zeilenweise festhält, wie sich die Werte von Variablen schrittweise verändern.

Um verschiedene Positionen in einem Programm zu benennen schreiben wir hinter jede Anweisung einen Kommentar mit einer fortlaufenden Nummer. Auch die Bedingung in bedingten Anweisungen und bedingten Schleifen benennen wir mit solchen sogenannten Programmpunkten.

Das folgende Programm zur Berechnung des Absolutbetrags des Wertes einer Variablen x ist mit Programmpunkten annotiert.

```
x = -4          #1
if x < 0 then   #2
  x = -1*x      #3
end
```

Die folgende Tabelle dokumentiert die Ausführung dieses Programms.

Programmpunkt (PP)	x	x < 0
#1	-4	
#2		true
#3	4	

Jede Zeile der Tabelle beschreibt Werte von Variablen oder Bedingungen an einem bestimmten Programmpunkt. Der Wert von x ändert sich zwei mal, der Wert der Bedingung x < 0 wird einmal ausgewertet.

Die Programmpunkte eines Programms brauchen nicht alle genau in ihrer textuellen Reihenfolge durchlaufen zu werden. Beim Programm zur Berechnung des Maximums zweier Zahlen wird zum Beispiel eine Zuweisung übersprungen.

```
x = 4           #1
y = 5           #2
if x > y then   #3
  z = x         #4
else
  z = y         #5
end
puts(z)         #6
```

Die folgende Tabelle dokumentiert die Ausführung dieses Programms.

PP	x	y	z	x > y	Ausgabe
#1	4				
#2		5			

PP	x	y	z	x > y	Ausgabe
#3				false	
#5			5		
#6					5

Hier wird der Programmpunkt #4 im `then`-Zweig der bedingten Anweisung übersprungen, weil die Bedingung `x > y` nicht erfüllt ist. Neben den verwendeten Variablen und Bedingungen dokumentiert diese Tabelle auch die Ausgaben mit `puts` im Terminal.

Interessant werden solche Tabellen besonders, wenn Anweisungen durch Schleifen wiederholt werden. Auch die Deklaration einer Zählschleife bekommt dabei eine eigene Nummer, um die Werte der Zählvariable zu protokollieren.

Hier ist ein Programm, angereichert mit Programmpunkten, zur Berechnung der Summe der ersten drei Zahlen.

```
sum = 0             #1
for i in 1..3 do    #2
  sum = sum + i     #3
end
puts(sum)           #4
```

Die folgende Tabelle protokolliert dessen Ausführung.

PP	sum	i	Ausgabe
#1	0		
#2		1	
#3	1		
#2		2	
#3	3		
#2		3	
#3	6		
#4			6

Hier werden die Programmpunkte #2 und #3 dreimal hintereinander durchlaufen, wobei die sich ändernden Werte der Variablen `i` und `sum` protokolliert werden.

Schließlich dokumentieren wir noch die Berechnung der Wurzel aus neun als Beispiel eines Programms mit bedingter Schleife.

```
n = 9                 #1
i = 0                 #2
q = 0                 #3
```

```
while q < n do      #4
  i = i + 1         #5
  q = q + 2*i - 1   #6
end
puts(i)             #7
```

Die folgende Tabelle zeigt, wie es zur der Ausgabe `3` am Ende des Programms kommt.

PP	n	i	q	q < n	Ausgabe
#1	9				
#2		0			
#3			0		
#4				true	
#5		1			
#6			1		
#4				true	
#5		2			
#6			4		
#4				true	
#5		3			
#6			9		
#4				false	
#7					3

Hier werden die Programmpunkte #4, #5 und #6 dreimal durchlaufen. Es wird deutlich, dass vor und nach jedem Schleifendurchlauf der Wert der Variablen `q` gleich dem Quadrat des Wertes von `i` ist. Eine solche Bedingung, die sich durch die Ausführung des Schleifenrumpfes nicht verändert, heißt Schleifen-Invariante. Sie hilft uns zu erkennen, dass die Ausgabe des Programms die Quadratwurzel von `q` ist. Ist `n` eine Quadratzahl, so ist bei Programmende `q` gleich `n`, die Ausgabe also die Quadratwurzel von `n`.

Quellen und Lesetipps

- Offizielle Ruby Dokumentation auf englisch und deutsch
- Edsger W. Dijkstra: A case against the GO TO statement
- Why's (Poignant) Guide to Ruby

4

Grundlegende Programmiertechniken

Programme folgen oft gewissen Entwurfsmustern oder Programmiertechniken. Solche Muster zu erkennen erfordert Erfahrung um sie als Gemeinsamkeiten vieler unterschiedlicher Programme zu entdecken. Wer sie erkennen kann, wird Programme, die bekannten Mustern folgen, schneller verstehen. Wer entscheiden kann, wann welche Techniken anwendbar sind, wird Programme schneller schreiben können. Im folgenden betrachten wir einige grundlegende Programmiertechniken, um die Programmierung, vor allem mit Schleifen, weiter zu vertiefen.

Aufzählen und Überprüfen

Ein Vorteil eines Computers gegenüber einem Menschen ist die Fähigkeit, viele Werte sehr schnell aufzählen und gewisse Eigenschaften für diese Werte testen zu können. Somit können viele Probleme, bei denen der Bereich der möglichen Lösungen endlich ist und aufgezählt werden kann, mit der Programmiertechnik Aufzählen und Überprüfen gelöst werden.

Als Beispiel für diese Technik betrachten wir die Berechnung des größten gemeinsamen Teilers zweier natürlicher Zahlen. Der größte gemeinsame Teiler zweier Zahlen wird z.B. beim Kürzen von Brüchen verwendet, wobei Zähler und Nenner durch ihren größten gemeinsamen Teiler dividiert werden.

Mathematisch kann der größte gemeinsame Teiler (ggT) wie folgt definiert werden. Für gegebene natürliche Zahlen $a, b \in \mathbb{N}$ ist $ggT(a, b) = c \in \mathbb{N}$ diejenige natürliche Zahl für die gilt: c teilt a ohne Rest, c teilt b ohne Rest und für alle weiteren Teiler d von a und b gilt $c > d$.

Als Beispiel betrachten wir folgende Zahlen.

- 21 hat die Teiler $1, 3, 7$ und 21.
- 18 hat die Teiler $1, 2, 3, 6, 9$ und 18.

Der größte gemeinsame Teiler von 21 und 18 ist also $ggT(21,18) = 3$.

Jede positive Zahl ist ein Teiler der Null. Für alle $a > 0$ gilt also $ggT(a,0) = ggT(0,a) = a$. Der Wert von $ggT(0,0)$ ist nicht definiert, da alle positiven Zahlen Teiler von 0 sind; es gibt also keinen größten gemeinsamen Teiler.

Für die Überprüfung, ob eine Zahl eine andere ohne Rest teilt kann der Modulo-Operator verwendet werden, welcher den Rest einer ganzzahligen Division liefert. Falls `a` und `b` ganzzahlige Werte sind, so liefert `%` den Rest der ganzzahligen Division von `a` durch `b`.

Wie können wir das Problem der Berechnung größter gemeinsamer Teiler algorithmisch lösen? Eine einfache Methode ist die Berechnung durch Aufzählen und Überprüfen.[1]

[1] Es gibt bessere Methoden zur Berechnung des ggT. Der Euklidische Algorithmus berechnet den ggT zweier Zahlen deutlich schneller als das folgende Programm.

Der ggT von a und b liegt sicherlich zwischen 1 und der kleineren der beiden Zahlen. Wir können also diese Werte der Reihe nach aufzählen und jeweils testen, ob die entsprechende Zahl beide gegebenen Zahlen ohne Rest teilt.

Für die Überprüfung, ob eine Zahl eine andere ohne Rest teilt kann der Modulo-Operator verwendet werden, welcher den Rest einer ganzzahligen Division liefert. Falls `a` und `b` ganzzahlige Werte sind, so liefert `%` den Rest der ganzzahligen Division von `a` durch `b`.

Das folgende Programm berechnet zunächst das Minimum `min` gegebener Zahlen `a` und `b` und sucht dann in einer Zähl-Schleife den größten gemeinsamen Teiler dieser Zahlen.

```
a = 4                            #1
b = 6                            #2

if a < b then                    #3
  min = a                        #4
else
  min = b                        #5
end

for i in 1..min do               #6
  if a%i == 0 && b%i == 0 then   #7
    ggT = i                      #8
  end
end

puts(ggT)                        #9
```

Wir verwenden eine Zähl-Schleife, da wir alle Werte zwischen `1` und `min` daraufhin testen wollen, ob sie ein Teiler von sowohl

`a` als auch `b` sind. Wir wissen also vorher, wieviele Schleifendurchläufe dafür gebraucht werden. Die Bedingung für den Test ist wegen der Präzedenzen der beteiligten Operatoren so geklammert: `((a%i) == 0) && ((b%i) == 0)`. Bei Programmende ist die größte Zahl `i`, die diese Bedingung erfüllt (also der ggT von `a` und `b`) in der Variablen `ggT` gespeichert.

Die folgende Tabelle dokumentiert die Ausführung dieses Programms.

PP	a	b	a < b	min	i	a%i == 0 && b%i == 0	ggT	Ausgabe
#1	4							
#2		6						
#3			true					
#4				4				
#6					1			
#7						true		
#8							1	
#6					2			
#7						true		
#8							2	
#6					3			
#7						false		
#6					4			
#7						false		
#9								2

Statt alle Zahlen zu durchlaufen, können wir auch von oben anfangen aufzuzählen. Diese Vorgehensweise hat den Vorteil, dass der erste gefundene gemeinsame Teiler auch der größte ist. Wir können dann die Schleife beenden, sobald wir einen gemeinsamen Teiler gefunden haben.

Da wir hierbei nicht wissen, wieviele Schleifendurchläufe gebraucht werden, verwenden wir zur Implementierung dieser Idee eine bedingte Schleife. Das folgende Programm bestimmt zunächst mit einer bedingten Verzweigung die kleinere der beiden Eingabezahlen und sucht dann mit einer bedingten Schleife abwärts nach dem größten gemeinsamen Teiler, der schließlich mit einer `puts`-Anweisung ausgegeben wird.

```
a = 4                                  #1
b = 6                                  #2

if a < b then                          #3
  ggT = a                              #4
else
  ggT = b                              #5
end
```

```
while a%ggT != 0 || b%ggT != 0 do   #6
  ggT = ggT - 1                     #7
end

puts(ggT)                           #8
```

Die folgende Tabelle dokumentiert die Ausführung dieses Programms.

PP	a	b	a < b	ggT	a%ggT != 0 || b%ggT != 0	Ausgabe
#1	4					
#2		6				
#3			true			
#4				4		
#6					true	
#7				3		
#6					true	
#7				2		
#6					false	
#8						2

Setzen wir eine der Zahlen `a` und `b` gleich `0`, liefert dieses Programm einen Laufzeitfehler wegen Division durch Null. Um dies zu verhindern müssen wir die Randfälle, in denen mindestens eine der Eingabezahlen Null ist, prüfen und unseren Algorithmus nur dann ausführen, wenn beide Zahlen ungleich Null sind.

```
a = 4
b = 6

if a == 0 && b == 0 then
  puts("nicht definiert")
else
  if a == 0 then
    puts(b)
  end
  if b == 0 then
    puts(a)
  end
  if a != 0 && b != 0 then
    if a < b then
      ggT = a
    else
      ggT = b
    end
```

```
    while a%ggT != 0 || b%ggT != 0 do
      ggT = ggT - 1
    end

    puts(ggT)
  end
end
```

Hier zeigt sich, dass es beim Testen von Programmen wichtig ist, Randfälle systematisch zu überprüfen. Manchmal wird ein Programm leider bei korrekter Behandlung der Randfälle wie hier etwas aufgebläht.

Teilen und Herrschen

Statt Lösungskandidaten der Reihe nach aufzuzählen, können wir einige Problem auch lösen, indem wir den durchsuchten Bereich geschickter eingrenzen. Das Verfahren Teile und Herrsche zerlegt ein Problem in, beispielsweise, zwei halb so große Teilprobleme, die dann mit der selben Technik gelöst werden können. Als Beispiel für ein solches Problem betrachten wir das Spiel Zahlenraten.

Eine Spielerin denkt sich eine Zahl zwischen 1 und 100 ohne sie zu verraten. Die Gegenspielerin muss die gedachte Zahl möglichst schnell erraten, wobei sie auf Rateversuche jedoch nur die Antworten "Ja, erraten.", "Nein, meine Zahl ist kleiner." oder "Nein, meine Zahl ist größer." erhält.

Natürlich können wir, um die gedachte Zahl zu erraten einfach alle Zahlen der Reihe nach abfragen, bis wir die richtige Zahl gefunden haben. Deutlich schneller gelangen wir jedoch ans Ziel, wenn wir den durchsuchten Bereich in jedem Schritt halbieren.

Das folgende Programm implementiert diese Idee.

```
min = 1
max = 100
geheim = 37

erraten = false
while !erraten do
  kandidat = (min + max) / 2
  puts("Ist die Zahl gleich " + kandidat.to_s + "?")

  if geheim == kandidat then
    puts("Ja, erraten.")
    erraten = true
  end
```

```
  if geheim < kandidat then
    puts("Nein, meine Zahl ist kleiner.")
    max = kandidat - 1
  end

  if geheim > kandidat then
    puts("Nein, meine Zahl ist größer.")
    min = kandidat + 1
  end
end
```

Hier wird der durchsuchte Bereich von `min` bis `max` in jedem Schleifendurchlauf halbiert. Wenn die Zahl erraten wurde, wird die Schleife durch die Zuweisung `erraten = true` beendet.

Die Ausgabe dieses Programms ist

```
Ist die Zahl gleich 50?
Nein, meine Zahl ist kleiner.
Ist die Zahl gleich 25?
Nein, meine Zahl ist größer.
Ist die Zahl gleich 37?
Ja, erraten.
```

Die gedachte Zahl wird mit dem Verfahren Teile und Herrsche in diesem Fall also nach drei Schritten gefunden. Der Algorithmus hat in diesem Fall Glück gehabt, weil er den Bereich garnicht bis zum Ende eingrenzen musste. Im schlimmsten Fall nähern sich `min` und `max` bei der Ausführung so weit an, dass sie gleich groß sind. In dem Fall ist das Problem dann aber einfach gelöst.

Vertiefung

Dieser Abschnitt beschreibt Programme, die die bisher behandelten Sprachmittel imperativer Programmiersprachen am Beispiel neuer Algorithmen vertiefen.

Geschachtelte Schleifen

Bisher kamen in den Schleifen-Rümpfen unserer Programme keine Schleifen vor. Insbesondere beim Aufzählen und Überprüfen kann es passieren, dass Schleifen geschachtelt werden, wenn der Test selbst eine Schleife verwendet oder mehrere Schleifen verwendet werden, um Kandidaten aufzuzählen.

Als Beispiel für ein Programm, dass Kandidaten mit Hilfe mehrerer geschachtelter Schleifen aufzählt, berechnen wir sogenannte Pythagoräische Tripel. Positive ganze Zahlen $a \leq b \leq c$ heißen Pythagoräisches Tripel, wenn $a^2 + b^2 = c^2$ gilt. Das folgende Programm listet alle solche Tripel aus Werten zwischen 1 und 20 auf.

```
n = 20

for a in 1..n do
  for b in a..n do
    for c in b..n do
      if a*a + b*b == c*c then
        puts(a.to_s + "," + b.to_s + "," + c.to_s)
      end
    end
  end
end
```

Hier besteht der Test aus einer einfachen Bedingung aber die Aufzählung geschieht mit Hilfe von drei geschachtelten Zählschleifen.

Die Ausgabe dieses Programms ist

```
3,4,5
5,12,13
6,8,10
8,15,17
9,12,15
12,16,20
```

Als Beispiel für die Programmiertechnik Aufzählen und Überprüfen, bei dem auch der Test eine Schleife verwendet, berechnen wir vollkommene Zahlen. Eine Zahl heißt vollkommen, wenn sie gleich der Summe aller ihrer Teiler ist, die kleiner sind als sie selbst. Das folgende Programm gibt alle vollkommenen Zahlen zwischen 1 und 1000 aus.

```
n = 1000

for i in 1..n do
  sum = 0
  for j in 1..i-1 do
    if i%j == 0 then
      sum = sum + j
    end
  end
  if i == sum then
```

```
    puts(i)
  end
end
```

Hier besteht der Test aus der Berechnung der Summe aller kleineren Teiler von `i` und dem anschließenden Vergleich dieser Summe mit `i`.

Die Ausgabe dieses Programms ist

```
6
28
496
```

Es scheint also sehr wenige vollkommene Zahlen zu geben.

Euklidischer Algorithmus

Der größte gemeinsame Teiler zweier Zahlen läßt sich mit dem Euklidischen Algorithmus schneller berechnen als bisher gesehen. Der Algorithmus wurde etwa 300 v. Chr. von Euklid beschrieben und ist einer der ältesten heute noch verwendeten Algorithmen. Der Algorithmus basiert auf der Idee, dass der größte gemeinsame Teiler zweier natürlicher Zahlen sich nicht ändert, wenn man die größere Zahl durch die Differenz der beiden Zahlen ersetzt. Es genügt also, den ggT dieser beiden neuen Zahlen zu berechnen, wodurch das Problem verkleinert wird.[2] Dieses Verfahren wird so lange fortgesetzt, bis beide Zahlen gleich groß sind. Sie entsprechen dann dem ggT der ursprünglichen Zahlen.

[2] Da das Problem wie beschrieben auf ein kleineres Problem zurück geführt wird, fassen einige den Euklidischen Algorithmus unter die Technik "Teile und Herrsche". Da das Ausgangsproblem allerdings nur auf *ein einziges* kleineres Problem zurück geführt wird, ist es fraglich, ob hier von "Teilen" die Rede sein kann.

Als Beispiel berechnen wir den ggT der Zahlen 49 und 21 anhand dieser Idee:

- Der ggT von 49 und 21 ist der ggT von 49-21 und 21.
- Der ggT von 28 und 21 ist der ggT von 28-21 und 21.
- Der ggT von 7 und 21 ist der ggT von 7 und 21-7.
- Der ggT von 7 und 14 ist der ggT von 7 und 14-7.
- Der ggT von 7 und 7 ist 7.

Also ist der ggT von 49 und 21 gleich 7.

Wir implementieren nun die Anwendung des Euklidischen Algorithmus auf 49 und 21 in Ruby. Zu Beginn des Programms weisen wir die Eingabezahlen zwei Variablen `a` und `b` zu. Anschließend weisen wir schrittweise der größeren der beiden Variablen die Differenz der gespeicherten Zahlen zu, bis beide Variablen, die gleiche Zahl enthalten. Da wir nicht wissen, wieviele Schritte dazu notwendig sind, verwenden wir eine bedingte Schleife.

```
a = 49             #1
b = 21             #2

while a != b do    #3
  if a > b then    #4
    a = a - b      #5
  else
    b = b - a      #6
  end
end

puts(a)            #7
```

Zur Veranschaulichung werten wir dieses Programm wie folgt tabellarisch aus.

PP	a	b	a != b	a > b	Ausgabe
1	49				
2		21			
3			true		
4				true	
5	28				
3			true		
4				true	
5	7				
3			true		
4				false	
6		14			
3			true		
4				false	
6		7			
3			false		
7					7

In diesem Beispiel wird deutlich, dass unter Umständen eine Variable mehrfach von der anderen abgezogen wird; nämlich solange das Ergebnis größer ist als die abgezogene Zahl. Im obigen Beispiel wird die 21 zunächst von 49 und dann von 28 abgezogen, bis das Ergebnis 7 ist. Dann wird die 7 zuerst von 21 und dann von 14 abgezogen, bis das Ergebnis 7 ist. Der beschriebene Prozess der wiederholten Subtraktion der 21 von 49 endet mit dem Rest der Division von 49 durch 21. Würden wir die 7 am Ende noch einmal von 7 abziehen, würde der Prozess der wiederholten Subtraktion der 7 von 21 ebenfalls mit dem Rest der Division von 21 durch 7 enden. Diese Idee können wir verwenden, um die Anzahl der Schleifen-Durchläufe bei der Berechnung des Euklidischen Algorithmus zu verringern, indem wir die Subtraktion durch den Modulo-Operator ersetzen.

```
a = 49                          #1
b = 21                          #2

while a != 0 && b != 0 do       #3
  if a < b then                 #4
    b = b % a                   #5
  else
    a = a % b                   #6
  end
end

puts(a+b)                       #7
```

Als Schleifenbedingung testen wir, dass keine der Eingabezahlen Null ist, um Division durch Null zu vermeiden. Die Bedingung `a != b` kann entfallen, da in dem Fall im nächsten Schleifendurchlauf `a = 0` gesetzt wird, wonach die Schleife endet und die Ausgabe-Anweisung `a+b`, also `b` ausgibt. Die folgende Tabelle dokumentiert die Ausführung dieses Programms.

PP	a	b	a!=0 && b!=0	a < b	Ausgabe
1	49				
2		21			
3			true		
4				false	
6	7				
3			true		
4				true	
5		0			
3			false		
7					7

Diese Implementierung verwendet nur noch halb so viele Schleifendurchläufe wie die vorherige. Außerdem wechselt der Test der Bedingung `a < b` in jedem Durchlauf seinen Wert.

Da der Divisionsrest immer kleiner ist als die Zahl, durch die geteilt wurde, ist der Vergleich innerhalb des Schleifen-Rumpfes, welche der beiden Variablen `a` und `b` größer ist, nicht mehr nötig. Stattdessen können wir die Rollen der Variablen in jedem Schritt vertauschen und den Algorithmus beenden, sobald der berechnete Divisionsrest Null ist. Das folgende Programm implementiert diese Idee.

```
a = 49              #1
b = 21              #2

while b != 0 do     #3
```

```
  x = b          #4
  b = a % b      #5
  a = x          #6
end

puts(a + b)      #7
```

Dieses Programm kommt ebenfalls mit der Hälfte der Schleifendurchläufe aus, wie die tabellarische Auswertung zeigt. Statt drei Vergleichen benötigen wir pro Schleifendurchlauf nur noch einen.

PP	a	b	b != 0	x	Ausgabe
1	49				
2		21			
3			true		
4				21	
5		7			
6	21				
3			true		
4				7	
5		0			
6	7				
3			false		
7					7

Im Allgemeinen lässt sich zeigen, dass diese Variante des Euklidischen Algorithmus höchstens 5 mal so viele Schritte benötigt, wie die Anzahl der Ziffern der kleineren Zahl. Der Beweis dieser Eigenschaft markierte 1844 den Beginn der Komplexitätstheorie, die heute als Teil der Theoretischen Informatik erforscht wird.

Quellen und Lesetipps

- Trial and Error
- Teile und Herrsche
- Euklidischer Algorithmus

5

Funktionen und Prozeduren

Bei der Einführung des Algorithmus-Begriffs haben wir diskutiert, dass die Benennung von Algorithmen ein wichtiges Mittel zur Abstraktion ist, welches erlaubt, einmal definierte Algorithmen wieder zu verwenden und dadurch komplexere Algorithmen auf Basis von einfacheren zu definieren. Zum Beispiel haben wir auf Basis eines Algorithmus `LINIE` einen Algorithmus `QUADRAT` definiert, der `LINIE` viermal verwendet hat. Ein weiterer wichtiger Abstraktionsmechanismus ist Parametrisierung, die es erlaubt statt eines konkreten Problems eine Klasse von Problemen mit einem einzigen Algorithmus zu lösen. So konnten wir zum Beispiel einen Algorithmus zum Zeichnen eines Kreises mit beliebigem Radius definieren.

Auch bei unseren Programmen wäre eine solche Strukturierung wünschenswert. Hierzu bieten Programmiersprachen Funktionen (mit Rückgabewert) und Prozeduren (ohne Rückgabewert). Funktionen dienen der Abstraktion von Ausdrücken, Prozeduren der Abstraktion von Anweisungen.

Abstraktion von Ausdrücken durch Funktionen

Einen Ausdruck zur Berechnung des Maximums zweier Zahlen können wir zum Beispiel wie folgt als Funktion abstrahieren.

```
def max(x,y)
  if x > y then
    z = x
  else
    z = y
  end
  return(z)
end
```

Das Schlüsselwort `def` leitet die Funktionsdefinition ein, `max` ist der Name der definierten Funktion und die Variablen `x` und

y heißen formale Parameter der Funktion max. Der sogenannte Funktionsrumpf enthält die bedingte Anweisung zur Berechnung des Maximums z der Werte von x und y. In der letzten Zeile wird mit Hilfe des Schlüsselwortes return, der Wert von z als Rückgabewert der Funktion max festgelegt.

Eine Rückgabe-Anweisung mittels return beendet die Ausführung des Funktionsrumpfes auch dann, wenn sie nicht an dessen Ende steht. Wir können deshalb die Funktion max auch etwas kürzer wie folgt definieren.

```
def max(x,y)
  if x > y then
    return(x)
  else
    return(y)
  end
end
```

Wenn wir diese Funktion in einem Ruby-Programm maxFun.rb speichern, können wir es wie folgt in irb einbinden und ausführen.

```
irb> load "maxFun.rb"
irb> max(2,3)
=> 3
```

Die load-Anweisung bewirkt also, dass max wie eine vordefinierte Funktion verwendet werden kann.

Um das Maximum dreier Zahlen zu berechnen, können wir nun statt einer geschachtelten bedingten Anweisung geschachtelte Funktions-Aufrufe verwenden.

```
irb> max(1,max(3,2))
=> 3
```

Bei der Auswertung eines Funktionsaufrufes werden zunächst die Argumente (auch aktuelle Parameter genannt) ausgewertet und dann in den Funktionsrumpf eingesetzt. Wir können die Auswertungsreihenfolge sichtbar machen, indem wir Ausgaben in den Funktionsrumpf einbauen.

```
def max(x,y)
  puts("Aufruf: max(" + x.to_s + "," + y.to_s + ")")

  if x > y then
    puts("Rückgabewert: " + x.to_s)
    return(x)
```

```
  else
    puts("Rückgabewert: " + y.to_s)
    return(y)
  end
end
```

Nachdem wir das Programm mit der `load`-Anweisung neu geladen haben, können wir den obigen Ausdruck mit den eingefügten Ausgaben auswerten.

```
irb> load "maxFun.rb"
irb> max(1,max(3,2))
Aufruf: max(3,2)
Rückgabewert: 3
Aufruf: max(1,3)
Rückgabewert: 3
=> 3
```

Hierbei erkennen wir, dass zunächst der Aufruf `max(2,3)` zu `3` ausgewertet wird. Danach wird dieses Ergebnis als Argument des äußeren Aufrufs von `max` verwendet. Der Aufruf `max(1,3)` wird dann zu `3` ausgewertet.

Ausgaben wie hier sind oft nützlich zur Fehlersuche in Programmen. Zugunsten einer Trennung von Ausdrücken und Anweisungen sollte aber in Funktionen in der Regel auf Ausgaben verzichtet werden.

Beachten Sie den Unterschied zwischen Ausgabe-Anweisungen zur Ausgabe eines Wertes im Terminal und Rückgabe-Anweisungen zur Festlegung des Rückgabewertes von Funktionen. Eine Ausgabe-Anweisung legt keinen Rückgabewert fest und eine Rückgabe-Anweisung erzeugt keine Ausgabe im Terminal.

Primzahltest

Als weiteres Beispiel für Abstraktion durch Funktionen betrachten wir die Aufzählung aller Primzahlen bis zu einer gegebenen Obergrenze. Wenn wir den Primzahltest als Funktion `prime?(n)` abstrahieren, können wir ihn in einer Zählschleife aufrufen, statt die Definition des Tests in die Zählschleife zu kopieren.

```
def prime?(n)
  teilbar = false
  k = 2
  while !teilbar && k*k <= n do
    teilbar = (n % k) == 0
```

```
    k = k + 1
  end

  return(n > 1 && !teilbar)
end

max = 100
for i in 2..max do
  if prime?(i) then
    puts(i)
  end
end
```

Die Funktion `prime?` liefert einen Wahrheitswert zurück und wird deshalb auch Prädikat genannt. Per Konvention, enden Namen von Prädikaten in Ruby oft mit einem Fragezeichen. Dies ist nicht vorgeschrieben, erhöht aber die Lesbarkeit.

Das definierte Prädikat `prime?` wird in einer Zählschleife nach seiner Definition aufgerufen.[1] Sein Ergebnis wird mit einer bedingten Anweisung überprüft um alle Primzahlen zwischen `2` und `max` auszugeben.

[1] Funktionen müssen in Ruby vor ihrem ersten Aufruf definiert werden.

Verarbeiten von Benutzereingaben

Bisher haben wir Beispiel-Eingaben immer direkt im Quelltext kodiert oder als Parameter von Funktionen oder Prozeduren in `irb` eingegeben. Im Folgenden diskutieren wir, wie wir Benutzereingaben im Terminal verarbeiten können.

Ruby stellt eine vordefinierte Funktion `gets` (für **get** __s__tring) zur Verfügung, mit deren Hilfe eine Zeile im Terminal eingelesen werden kann. Bei einem Aufruf von `gets()` wird die Abarbeitung des Programms so lange angehalten, bis eine Zeile (abgeschlossen mit der Enter-Taste) im Terminal eingegeben wurde. Das Ergebnis von `gets` ist die eingegebene Zeichenkette.

Die folgenden Aufrufe in `irb` zeigen von `gets` gelieferte Ergebnisse.

```
irb> gets()
Hallo
=> "Hallo\n"
irb> gets()
Dies ist ein ganzer Satz in einer Zeile.
=> "Dies ist ein ganzer Satz in einer Zeile.\n"
```

Nach dem Aufruf von `gets()` wartet `irb` auf eine Benutzereingabe. Nachdem wir etwas eingeben und die Enter-Taste drücken, wird die Eingabe als Zeichenkette zurückgegeben.[2]

[2] Wie wir sehen wird auch das Zeilenende-Zeichen `"\n"` im Ergebnis zurückgegeben. Um die Eingabe ohne Zeilenende-Zeichen zu erhalten, können wir `gets().chop` statt `gets` aufrufen.

Wir können `gets` wie folgt verwenden, um die Eingaben für die von uns definierte Funktion `max` im Terminal einzulesen.

```
puts("Gib eine Zahl ein.")
a = gets().to_i

puts("Gib noch eine Zahl ein.")
b = gets().to_i

puts("Die größere von beiden ist " + max(a,b).to_s + ".")
```

Wenn wir dieses Programm ausführen, werden wir zunächst nach zwei Zahlen gefragt und dann wird die größere von beiden ausgegeben:

```
Gib eine Zahl ein.
3
Gib noch eine Zahl ein.
2
Die größere von beiden ist 3.
```

Zum Einlesen der Werte der Variablen `a` und `b` verwenden wir `to_i` um den Rückgabewert von `gets` in eine Zahl umzuwandeln. Die eingelesenen Zahlen reichen wir als Argumente an die Funktion `max` weiter, deren Ergebnis wir mit `to_s` in eine Zeichenkette umwandeln um es auszugeben.

In diesem Programm gibt es ein wiederkehrendes Muster, das wir als Funktion abstrahieren können. Wir lesen zweimal eine Benutzereingabe ein, nachdem wir eine Eingabeaufforderung im Terminal ausgeben. Die Funktion `eingabe` erledigt das für beliebige Eingabeaufforderungen.

```
def eingabe(aufforderung)
  puts(aufforderung)
  return(gets().chop)
end
```

Mit ihrer Hilfe können wir unser Programm wie folgt vereinfachen.

```
a = eingabe("Gib eine Zahl ein.").to_i
b = eingabe("Gib noch eine Zahl ein.").to_i
puts("Die größere von beiden ist " + max(a,b).to_s + ".")
```

Wir können Benutzereingaben auch in einer Schleife einlesen, um interaktive Programme zu schreiben, die mit ihren Benutzern kommunizieren. Das folgende Programm fragt zum Beispiel so lange nach Eingaben, wie positive Zahlen eingegeben werden, und gibt dann aus, ob es sich bei der eingegebenen Zahl um eine Primzahl handelt.

```
s = eingabe("Gib eine Zahl ein.")
while s != "quit" do
  n = s.to_i
  if prime?(n) then
    puts(n.to_s + " ist eine Primzahl.")
  else
    puts(n.to_s + " ist keine Primzahl.")
  end
  s = eingabe("Gib noch eine Zahl ein.")
end
```

Hier ist eine Beispiel-Interaktion mit diesem Programm.

```
Gib eine Zahl ein.
17
17 ist eine Primzahl.
Gib noch eine Zahl ein.
21
21 ist keine Primzahl.
Gib noch eine Zahl ein.
quit
```

Nach Eingabe von `quit` wird die Schleife beendet, und es werden keine weiteren Fragen mehr gestellt.

Zahlenraten mit Benutzereingabe

Wir können nun auch unser Programm zum Zahlenraten so abwandeln, dass es eine vom Benutzer gedachte Zahl errät.

```
min = 1
max = 100

erraten = false

while !erraten do

  if min == max then
    puts("Die Zahl ist " + min.to_s + ".")
    erraten = true
  else
    kandidat = (min + max) / 2
    antwort = eingabe("Ist die Zahl " + kandidat.to_s + "?")

    if antwort == "=" then
     erraten = true
    end
```

```
    if antwort == "<" then
      max = kandidat - 1
    end

    if antwort == ">" then
      min = kandidat + 1
    end
  end
end
```

Hier ist eine Beispiel-Interaktion mit diesem Programm.

```
Ist die Zahl 50?
<
Ist die Zahl 25?
>
Ist die Zahl 37?
>
Ist die Zahl 43?
<
Ist die Zahl 40?
>
Ist die Zahl 41?
>
Die Zahl ist 42.
```

Abstraktion von Anweisungen durch Prozeduren

Das Programm zur Ausgabe aller Primzahlen bis zu einer Obergrenze legt die Obergrenze im Programmtext fest. Statt alle Primzahlen bis zu einer konkreten Obergrenze auszugeben, können wir auch ein Programm zur Ausgabe aller Primzahlen bis zu einer beliebigen Obergrenze schreiben. Dazu abstrahieren wir die Zählschleife mit Hilfe einer Prozedur `primesUpTo` mit einem Parameter `max`.

```
def primesUpTo(max)
  for i in 2..max do
    if prime?(i) then
      puts(i)
    end
  end
end
```

Prozedur-Aufrufe

Nun können wir, zum Beispiel in `irb`, die Anweisungen `primesUpTo(100)` und `primesUpTo(1000)` ausführen, um alle Primzahlen kleiner als 100 bzw. 1000 im Terminal auszugeben.

Prozeduren haben keinen Rückgabewert, enthalten also keine `return`-Anweisung. Sie können verwendet werden um mit `puts`-Anweisungen komplexe Ausgaben im Terminal zu erzeugen.

Zum Beispiel gibt die folgende Prozedur den Umriss eines Quadrates aus Sternchen im Terminal aus.

```
def putQuadrat(size)
  line = ""
  for i in 1..size do
    line = line + "*"
  end

  inside = ""
  for i in 1..size-2 do
    inside = inside + " "
  end

  puts(line)
  for i in 1..size-2 do
    puts("*" + inside + "*")
  end
  puts(line)
end
```

Eine mögliche Ausgabe des Programms sieht wie folgt aus.

```
****
*  *
*  *
****
```

Da wir das Quadrat zeilenweise ausgeben müssen, berechnen wir zunächst den oberen Rand als Zeile aus Sternchen gegebener Länge und speichern ihn in der Variable `line`. Danach berechnen wir das Innere als um zwei Zeichen kürzere Zeile `inside` aus Leerzeichen. Im Anschluss geben wir den oberen Rand gefolgt von Zeilen, die das Innere mit Sternchen umranden aus. Schließlich geben wir noch einmal `line` als unteren Rand aus.

Hier sind zwei Beispielausgaben dieser Prozedur in `irb`.

```
irb> putQuadrat(3)
***
* *
***
irb> putQuadrat(4)
****
```

```
*  *
*  *
****
```

Bei der Definition der Prozedur `putQuadrat` fällt eine Ähnlichkeit des Codes zur Berechnung der oberen und unteren Zeile sowie des inneren des Quadrates auf. Beide Male wird eine gegebene Zeichenkette eine bestimmte Anzahl oft wiederholt.

Wir können unser Programm vereinfachen, indem wir diese Berechnung als Funktion abstrahieren und dann innerhalb von `putQuadrat` verwenden.

```
def repeat(times,string)
  result = ""
  for i in 1..times do
    result = result + string
  end
  return(result)
end

def putQuadrat(size)
  puts(repeat(size,"*"))
  puts(repeat(size-2,"*" + repeat(size-2," ") + "*\n"))
  puts(repeat(size,"*"))
end
```

Der Rückgabewert der Funktion `repeat` ist eine Zeichenkette. Innerhalb der Prozedur `putQuadrat` werden verschiedene solcher Zeichenketten berechnet und mit `puts`-Anweisungen im Terminal ausgegeben.

Exkurs: Programmierung mit Zeichenketten

In diesem Einschub behandeln wir überblicksartig einige Möglichkeiten zur Programmierung mit Zeichenketten, auch Strings genannt. Um den Umgang mit Kontrollstrukturen zu vertiefen, werden wir einige Funktionen definieren, die Strings als Argumente erwarten.

Wir haben Zeichenketten bereits bit dem `+`-Operator aneinander gehängt. In Ruby können wir Zeichenketten auch mit Zahlen multiplizieren. Dabei wird wie bei der `repeat`-Funktion aus dem vorigen Abschnitt eine Zeichenkette eine gegebene Anzahl oft wiederholt.

```
irb> "*" * 5
=> "*****"
```

Wenn wir die Parameter vertauschen, liefert Ruby allerdings eine Fehlermeldung.

```
irb> 5 * "*"
TypeError: String can't be coerced into Fixnum
```

Zeilenumbrüche und Tabulatoren können als `\n` bzw. `\t` notiert werden. Der folgende Aufruf demonstriert die Verwendung dieser Steuerzeichen.

```
irb> puts(("*\t" * 5 + "\n") * 5)
*       *       *       *       *
*       *       *       *       *
*       *       *       *       *
*       *       *       *       *
*       *       *       *       *
```

Beim Einlesen von Zeichenketten im Terminal haben wir bereits `.chop` verwendet, um das letzte Zeichen des Rückgabewertes von `gets()` (also das Zeilenende-Zeichen) abzuschneiden. Um alle sogenannten whitespaces (also Leerzeichen, Tabulator-Zeichen, Zeilenende-Zeichen usw.) am Anfang und am Ende einer Zeichenkette zu entfernen, können wir `.strip` verwenden.

```
irb> " \t a  b c \n ".strip
=> "a  b c"
```

Teilstrings können durch Angabe eines Index und einer Länge in eckigen Klammern extrahiert werden. Die folgenden Aufrufe demonstrieren dies.

```
irb> "Hallo Welt!"[0,5]
=> "Hallo"
irb> "Hallo Welt!"[6,4]
=> "Welt"
irb> "Hallo Welt!"[10,10]
=> "!"
```

Wenn der String nicht genügend Zeichen enthält, ist der dabei zurückgelieferte Teilstring also kürzer als die angegebene Länge.

Wir können nun eine Funktion schreiben, die zählt, wie oft ein gegebenes Zeichen in einer Zeichenkette vorkommt.

```
def countChar(text,char)
  count = 0
  for index in 0..text.size-1 do
```

```
    if text[index,1] == char then
      count = count + 1
    end
  end
  return count
end
```

Hierzu durchlaufen wir in einer Zählschleife alle Zeichen der Zeichenkette `text` und erhöhen den Zähler `count`, wenn wir das gesuchte Zeichen finden. Die Zahlvariable `index` zählt die Positionen dabei mit Null beginnend durch.

Hier sind einige Beispielaufrufe.

```
irb> countChar("Hallo Welt!","l")
=> 3
irb> countChar("Hallo Welt!"," ")
=> 1
irb> countChar("Hallo Welt!","x")
=> 0
```

Wenn wir nur daran interessiert sind, ob das Zeichen enthalten ist, aber nicht daran wie oft, können wir stattdessen eine `while`-Schleife verwenden, um die Suche bei Erfolg vorzeitig zu beenden.

```
def containsChar?(text,char)
  found = false
  index = 0
  while !found && index < text.size do
    if text[index,1] == char then
      found = true
    end
    index = index + 1
  end
  return found
end
```

Das aktuelle Zeichen selektieren wir dabei wieder anhand einer Zählvariable `index` als Teilstring der Länge eins.

Diese Funktion können wir wie folgt verwenden.

```
irb> containsChar?("Hallo Welt!","l")
=> true
irb> containsChar?("Hallo Welt!"," ")
=> true
irb> containsChar?("Hallo Welt!","x")
=> false
```

Wir können auf diese Weise auch nach Teilstrings beliebiger Länge suchen und müssen dazu nur die selektierte Länge an den gesuchten String anpassen.

```
def containsString?(text,str)
  found = false
  index = 0
  while !found && index < text.size do
    if text[index,str.size] == str then
      found = true
    end
    index = index + 1
  end
  return(found)
end
```

Zur Illustration dienen wieder einige Beispielaufrufe.

```
irb> containsString?("Hallo Welt!","Hallo")
=> true
irb> containsString?("Hallo Welt!","Welt!")
=> true
irb> containsString?("Hallo Welt!","welt!")
=> false
```

Es wird also Groß- und Kleinschreibung unterschieden.

Schließlich können wir dieses Programm noch so abwandeln, dass es den ersten Index zurück gibt, an dem der gesuchte String gefunden wurde (bzw. eine Fehlermeldung ausgibt, falls er nicht gefunden wird).

```
def indexOf(text,str)
  found = false
  index = 0
  while !found && index < text.size do
    if text[index,str.size] == str then
      found = true
    end
    index = index + 1
  end
  if found then
    return (index-1)
  else
    puts("'" + str + "' kommt in '" + text + "' nicht vor.")
  end
end
```

Hierbei erniedrigen wir den Wert von `index`, bevor wir ihn zurück liefern, da er bei erfolgreicher Suche am Ende des Schleifenrumpfes einmal zu oft erhöht wird.

Hier sind Beispielaufrufe zur Illustration.

```
irb> indexOf("Hallo Welt!","Hallo")
=> 0
irb> indexOf("Hallo Welt!","Welt")
=> 6
irb> indexOf("Hallo Welt!","welt")
'welt' kommt in 'Hallo Welt!' nicht vor.
```

Mit der vordefinierten Funktion `IO.read` können wir den Inhalt von Textdateien als Zeichenketten einlesen. Dies ermöglicht es uns, auch in größeren Texten, zum Beispiel in unserem Programm, nach Zeichenketten zu suchen.

```
irb> source = IO.read("strings.rb")
irb> countChar(source,"?")
=> 2
irb> indexOf(source,"def containsString?")
=> 371
```

6

Programmierung mit Arrays

Bisher haben wir die folgenden Arten von Werten kennen gelernt: Zahlen mit oder ohne Komma, Wahrheitswerte und Zeichenketten. Gängige Programmiersprachen erlauben auch den Umgang mit komplexeren Werten, die einfachere Werte zusammen fassen. Komplexe Daten aus einfachen zusammen zu bauen und über festgelegte Funktionen und Prozeduren zu manipulieren, ist ein wichtiges Mittel zur Abstraktion in Programmen, das mit **Datenabstraktion** bezeichnet wird.

Eine Möglichkeit, mehrere Werte zu einem zusammenzufassen, ist durch sogenannte **Felder** oder **Arrays** gegeben. In Ruby werden Arrays durch eckige Klammern notiert, zwischen die die in Ihnen enthaltenen Werte durch Kommata getrennt geschrieben werden. Die folgenden Beispiele in `irb` demonstrieren den Umgang mit Arrays.

```
irb> [1,2,3]
=> [1, 2, 3]
irb> [1,2] + [3]
=> [1, 2, 3]
irb> a = [1,2,3]
=> [1, 2, 3]
irb> a[0]
=> 1
irb> a[1]
=> 2
irb> a[2]
=> 3
irb> for i in 0..a.size-1 do
irb*   puts(a[i])
irb> end
1
2
3
```

Wir können also mehrere Zahlen in einem Array zusammenfassen und ähnlich wie bei Zeichenketten Arrays mit dem

+-Operator verketten und auf einzelne gespeicherte Elemente über einen Index zugreifen.

Wir können auch andere Werte als Zahlen in Arrays speichern.

```
irb> strings = ["Hallo", "Welt"]
=> ["Hallo", "Welt"]
irb> bools = [true, false, true]
=> [true, false, true]
irb> arrays = [[1,2,3], strings, bools, []]
=> [[1, 2, 3], ["Hallo", "Welt"], [true, false, true], []]
irb> for i in 0..arrays.size-1 do
irb*   puts(arrays[i].size)
irb> end
3
2
3
0
```

Das letzte Beispiel zeigt, dass auch Arrays selbst wieder Elemente von Arrays sein können. Das letzte Element des definierten Feldes `array` ist dabei ein leeres Feld, also eines ohne Einträge.

Wir können Arrays verwenden, um zu berechnende Funktionswerte zum schnelleren Zugriff zu speichern. Zum Beispiel können wir ein Feld `factorials` anlegen, das alle Fakultäten von 0 bis 10 enthält:

```
factorials = [1]

for i in 1..10 do
  factorials[i] = i * factorials[i-1]
end
```

Der Rumpf der Zählschleife enthält hier ein sogenanntes Array-Update, mit dem der Wert an einem angegebenen Index überschrieben wird. Array-Updates ähneln Zuweisungen, allerdings steht bei ihnen links vom Gleichheitszeichen keine Variable sondern es wird eine durch einen Index beschriebene Position in einem Array referenziert.

Nach Ausführung des obigen Programms können wir die gespeicherten Fakultäten in dem Feld `factorials` nachschlagen, statt sie immer wieder neu zu berechnen. Falls wir mehrfach auf die selben Fakultäten zugreifen wollen, können wir deren wiederholte Berechnung also auf Kosten eines höheren Speicherbedarfs einsparen.

Suchen in Arrays

Die folgende Funktion sucht ein gegebenes Element `x` in einem Array `a` und gibt `true` aus, falls `x` in `a` enthalten ist, und sonst `false`.

```
def contains?(a,x)
  found = false                #1
  for i in 0..a.size-1 do      #2
    if x == a[i] then          #3
      found = true             #4
    end
  end

  return found                 #5
end
```

Die folgende Programmtabelle dokumentiert die Ausführung dieses Programms für die Argumente `a = [1,2,3,4,5]` und `x = 3`. Wir verzichten dabei auf die Angabe der Werte für `a` und `x`, die sich während der Ausführung nicht ändern.

PP	found	i	x == a[i]	Rückgabewert
#1	false			
#2		0		
#3			false	
#2		1		
#3			false	
#2		2		
#3			true	
#4	true			
#2		3		
#3			false	
#2		4		
#3			false	
#5				true

Wie wir sehen, durchläuft das Programm das gesamte Feld auch dann, wenn das Element schon gefunden wurde. Mit einer `while`-Schleife können wir erreichen, dass die Suche in diesem Fall beendet wird.

```
def contains?(a,x)
  found = false                      #1
  i = 0                              #2
  while !found && i < a.size do      #3
    if x == a[i] then                #4
```

```
      found = true                      #5
    end
    i = i + 1                           #6
  end

  return found                          #7
end
```

Die folgende Programmtabelle dokumentiert die Ausführung dieses Programms für die selbe Eingabe.

PP	found	i	!found && i < a.size	x == a[i]	Rückgabewert
#1	false				
#2		0			
#3			true		
#4				false	
#6		1			
#3			true		
#4				false	
#6		2			
#3			true		
#4				true	
#5	true				
#6		3			
#3			false		
#7					true

Wie wir sehen, bricht das Programm nun ab, wenn das gesuchte Element gefunden wurde.

7

Rekursion

Funktionen und Prozeduren abstrahieren komplexe Ausdrücke beziehungsweise Anweisungsfolgen und erlauben es dadurch, Algorithmen zu implementieren und wiederzuverwenden. Komplexe Algorithmen können so auf Basis einfacherer Algorithmen implementiert werden, indem erstere letztere in Ihrer Definition verwenden.

Bei der Definition von Funktionen und Prozeduren können jedoch nicht nur bereits definierte (potentiell einfachere) Funktionen und Prozeduren verwendet werden sondern auch die gerade definierte Funktion oder Prozedur selbst. Der Rückgriff auf die eigene Definition in deren Implementierung heißt Rekursion.

Rekursive Funktionen

In der Mathematik ist die Verwendung von Rekursion zur Definition von Funktionen Gang und Gäbe. Eine typische Definition der Fakultätsfunktion sieht zum Beispiel so aus.

```
n! = 1          falls n = 1
n! = n * (n-1)! sonst
```

Diese Definition können wir mit Hilfe einer bedingten Verzweigung direkt nach Ruby übersetzen.

```
def factorial(n)
  if n == 1 then
    return 1
  else
    return (n * factorial(n-1))
  end
end
```

Falls der Parameter n gleich 1 ist, ist das Ergebnis ebenfalls 1. Falls nicht, wird das Ergebnis mit Hilfe eines rekursiven Aufrufs berechnet.

Rekursive Funktionen sind oft auf diese Weise strukturiert. Mit einer bedingten Verzweigung wird eine sogenannte Abbruchbedingung geprüft, die darüber entscheidet, ob die Berechnung beendet wird oder weiter geht. Im Fall, dass die Berechnung weiter geht folgt ein rekursiver Aufruf, im Abbruchfall nicht. Falls die Abbruchbedingung nie erfüllt ist, terminiert die Berechnung nicht, genau wie bei einer bedingten Schleife, deren Schleifenbedingung immer erfüllt ist.

Programmtabellen sind nicht geeignet um die Auswertung rekursiver Funktionen zu veranschaulichen, da deren Parameter-Variablen für jeden Aufruf unterschiedliche Werte haben können. Unterschiedliche Variablen mit dem selben Namen in einer Tabelle zu verwalten wird schnell unübersichtlich.

Statt mit einer Programmtabelle können wir die Auswertung rekursiver Funktionen wie hier am Beispiel der Fakultätsfunktion demonstriert veranschaulichen.

```
factorial(4)
Die Abbruchbedingung ist nicht erfüllt.
Das Ergebnis von factorial(4) ist 4 * factorial(3)
Es muss zunächst factorial(3) ausgewertet werden.
    factorial(3)
    Die Abbruchbedingung ist nicht erfüllt.
    Das Ergebnis von factorial(3) ist 3 * factorial(2).
    Es muss zunächst factorial(2) ausgewertet werden.
        factorial(2)
        Die Abbruchbedingung ist nicht erfüllt.
        Das Ergebnis von factorial(2) ist 2 * factorial(1).
        Es muss zunächst factorial(1) ausgewertet werden.
            factorial(1)
            Die Abbruchbedingung ist erfüllt.
            Das Ergebnis von factorial(1) ist 1.
        Das Ergebnis von factorial(2) ist also 2 * 1 = 2
    Das Ergebnis von factorial(3) ist also 3 * 2 = 6
Das Ergbenis von factorial(4) ist also 4 * 6 = 24
```

Statt einer Funktion, die sich selbst aufruft, können wir auch Gruppen mehrerer rekursiver Funktionen definieren, die sich gegenseitig aufrufen. Die folgenden Definitionen illustrieren diese Technik.

```
def is_even?(n)
    if n == 0 then
        return true
    else
```

```
        return is_odd?(n-1)
    end
end

def is_odd?(n)
    if n == 0 then
        return false
    else
        return is_even?(n-1)
    end
end
```

Die Funktion `is_even?` liefert `true` oder `false` zurück, je nachdem ob die gegebene Zahl `n` gerade ist oder nicht. Sie verwendet dazu im rekursiven Fall die Funktion `is_odd?`, die ihrerseits `is_even?` im rekursiven Fall verwendet. Die Abbruchbedingung beider Funktionen testet, ob das Argument gleich Null ist. Für negative Eingaben definieren diese Funktionen deshalb nicht, was es heißt, gerade oder ungerade zu sein.

Rekursive Prozeduren

Nicht nur Funktionen sondern auch Prozeduren können rekursiv definiert werden. Als Beispiel betrachten wir die folgende Prozedur.

```
def countdown(n)
    puts(n)
    if n > 0 then
        countdown(n-1)
    end
end
```

Die Prozedur `countdown` erwartet eine ganze Zahl als Argument und gibt zunächst die übergebene Zahl aus. Ist sie größer als Null, folgt ein rekursiver Aufruf mit der nächstkleineren ganzen Zahl. Dadurch werden bei Übergabe einer positiven Ganzzahl nacheinander alle ganzen Zahlen von der übergebenen Zahl bis Null ausgegeben, wie der folgende Aufruf zeigt.

```
irb> countdown(5)
5
4
3
2
1
0
```

Rekursive Aufrufe müssen nicht immer am Ende einer Definition stehen. Die folgende Prozedur ruft sich selbst auf, bevor ein Wert ausgegeben wird.

```
def countupto(n)
    if n > 0 then
        countupto(n-1)
        puts(n)
    end
end
```

Wie der Name suggeriert, zählt diese Prozedur aufwärts, denn bevor die übergebene Zahl ausgegeben wird, werden rekursiv alle natürlichen Zahlen bis zur um eins kleineren als die übergebene ausgegeben.

```
> countupto(5)
1
2
3
4
5
```

Im Rahmen der Übung sollen Sie damit experimentieren, was passiert, wenn eine Prozedur sich in ihrem Rumpf mehr als einmal selbst aufruft.

Rekursion und Schleifen

Wir haben gesehen, dass mit Hilfe von Rekursion, wie mit bedingten Schleifen, nicht terminierende Berechnungen beschrieben werden können. In der Tat sind Rekursion und bedingte Schleifen gleichmächtig, das heißt jedes Programm mit bedingter Schleife kann in eines übersetzt werden, dass statt dieser Rekursion verwendet und umgekehrt. Im folgenden übersetzen wir beispielhaft eine Funktion mit bedingter Schleife in eine rekursive Funktion ohne Schleifen. Die umgekehrte Übersetzung rekursiver Funktionen in Schleifen betrachten wir nicht.

Die folgende Funktion `factLoop` berechnet die Fakultät des Parameters `n` mit Hilfe einer bedingten Schleife.

```
def factLoop(n)
  f = 1
  i = 1
  while i <= n do
    f = f * i
    i = i + 1
```

```
  end
  return f
end
```

Wir können diese Funktion systematisch in eine rekursive Funktion übersetzen. Dazu definieren wir eine Funktion `factRec`, die neben dem Parameter `n` auch noch weitere Parameter für alle vor der Schleife definierten Variablen hat.

```
def factRec(n,f,i)
  if i <= n then
    f = f * i
    i = i + 1
    return factRec(n,f,i)
  else
    return f
  end
end
```

Im Rumpf dieser Funktion testen wir die Schleifenbedingung mit einer bedingten Verzweigung. Ist sie erfüllt, so führen wir den Schleifenrumpf einmal aus und rufen die Funktion dann rekursiv mit geänderten Parametern auf. Ist die Schleifenbedingung nicht erfüllt, bricht die Rekursion ab und wir führen die Anweisungen aus, die nach der ursprünglichen Schleife kommen, also `return f`.

Zur Initialisierung der zusätzlichen Parameter führen wir die Anweisungen vor der Schleife aus und rufen dann die Funktion `factRec` auf.

```
def fact(n)
  f = 1
  i = 1
  return factRec(n,f,i)
end
```

Wir veranschaulichen die Auswertung des Aufrufs `fact(4)` analog zur Auswertung von `factorial(4)`.

```
fact(4)
Das Ergebnis von fact(4) ist factRec(4,1,1)
factRec(4,1,1)
Die Schleifenbedingung ist erfüllt.
Das Ergebnis von factRec(4,1,1) ist factRec(4,1*1,1+1).
factRec(4,1,2)
Die Schleifenbedingung ist erfüllt.
Das Ergebnis von factRec(4,1,2) ist factRec(4,1*2,2+1).
factRec(4,2,3)
```

```
Die Schleifenbedingung ist erfüllt.
Das Ergebnis von factRec(4,2,3) ist factRec(4,2*3,3+1).
factRec(4,6,4)
Die Schleifenbedingung ist erfüllt.
Das Ergebnis von factRec(4,6,4) ist factRec(4,6*4,4+1).
factRec(4,24,5)
Die Schleifenbedingung ist nicht erfüllt.
Das Ergebnis von factRec(4,24,5) ist 24.
Das Ergebnis von fact(4) ist also 24.
```

Diesmal notieren wir die rekursiven Aufrufe nicht als verschachtelte Nebenrechnungen, da deren Ergebnisse nicht mehr weiter verrechnet sondern direkt als Ergebnis verwendet werden. Die Zwischenergebnisse werden im zweiten Parameter `f` von `factRec` mitgeführt, die Zählvariable im dritten Parameter `i`.

Die systematische Übersetzung der Fakultätsberechnung mit einer bedingten Schleife in eine rekursive Funktion führt also zu einer alternativen Implementierung, die sich von unserer ursprünglichen rekursiven Fakultätsfunktion sowohl syntaktisch als auch bezüglich ihrer Ausführung unterscheidet.

Die umgekehrte Übersetzung rekursiver Funktionen mit Hilfe von Schleifen ist nicht trivial. Das im folgenden Abschnitt gezeigte Programm, lässt sich nicht so einfach ohne Rekursion ausdrücken (möglich ist es aber).

Türme von Hanoi

Als weiteres Beispiel einer rekursiven Prozedur lösen wir das Problem der Türme von Hanoi, das Wikipedia so beschreibt:

> Das Spiel besteht aus drei gleichgroßen Stäben A, B und C, auf die mehrere gelochte Scheiben gelegt werden, alle verschieden groß. Zu Beginn liegen alle Scheiben auf Stab A, der Größe nach geordnet, mit der größten Scheibe unten und der kleinsten oben. Ziel des Spiels ist es, den kompletten Scheiben-Stapel von A nach C zu versetzen. Bei jedem Zug darf die oberste Scheibe eines beliebigen Stabes auf einen der beiden anderen Stäbe gelegt werden, vorausgesetzt, dort liegt nicht schon eine kleinere Scheibe. Folglich sind zu jedem Zeitpunkt des Spieles die Scheiben auf jedem Feld der Größe nach geordnet.

Die folgende Prozedur ist parametrisiert über die initiale Anzahl `n` der zu versetzenden Scheiben und gibt Anweisungen der Form

```
Lege eine Scheibe von X nach Y.
```

aus, wobei für `X` und `Y` jeweils einer der Stäbe `A`, `B` oder `C` eingesetzt wird.

```
def hanoi(n)
  hanoiRec(n,"A","B","C")
end
```

Um `n` Scheiben von Stab A über Stab B zu Stab C zu versetzen, können wir, falls `n` größer als `1` ist, zunächst `n-1` Scheiben von A über C nach B versetzen, dann die größte Scheibe von A nach C legen und schließlich die `n-1` Scheiben von Stab B über A nach C versetzen. Die Prozedur `hanoiRec` implementiert diese Idee für beliebige Start, Hilfs- und Ziel-Stäbe.

```
def hanoiRec(n, from, over, to)
  if n == 1 then
    puts("Lege eine Scheibe von " + from + " nach " + to + ".")
  else
    hanoiRec(n-1, from, to, over)
    puts("Lege eine Scheibe von " + from + " nach " + to + ".")
    hanoiRec(n-1, over, from, to)
  end
end
```

Im Folgenden veranschaulichen wir die Ausführung des Aufrufs `hanoi(3)`.

```
hanoi(3)
hanoiRec(3,"A","B","C")
Die Abbruchbedingung ist nicht erfüllt.
    hanoiRec(2,"A","C","B")
    Die Abbruchbedingung ist nicht erfüllt.
        hanoiRec(1,"A","B","C")
        Die Abbruchbedingung ist erfüllt.
        puts("Lege eine Scheibe von A nach C.")
    puts("Lege eine Scheibe von A nach B.")
        hanoiRec(1,"C","A","B")
        Die Abbruchbedingung ist erfüllt.
        puts("Lege eine Scheibe von C nach B.")
puts("Lege eine Scheibe von A nach C.")
    hanoiRec(2,"B","A","C")
    Die Abbruchbedingung ist nicht erfüllt.
        hanoiRec(1,"B","C","A")
        Die Abbruchbedingung ist erfüllt.
        puts("Lege eine Scheibe von B nach A.")
    puts("Lege eine Scheibe von B nach C.")
        hanoiRec(1,"A","B","C")
        Die Abbruchbedingung ist erfüllt.
        puts("Lege eine Scheibe von A nach C.")
```

Die gesamte Ausgabe dieses Aufrufs ist also die folgende.

```
Lege eine Scheibe von A nach C.
Lege eine Scheibe von A nach B.
Lege eine Scheibe von C nach B.
Lege eine Scheibe von A nach C.
Lege eine Scheibe von B nach A.
Lege eine Scheibe von B nach C.
Lege eine Scheibe von A nach C.
```

Im einzelnen nachzuvollziehen, welche Ausgabe in rekursiven Prozeduren wann erzeugt wird, ist oft trickreich, besonders dann, wenn Ausgaben *nach* rekursiven Aufrufen erfolgen. Häufig ist es jedoch garnicht notwendig die Ausführung rekursiver Programme im Detail nachzuvollziehen. Es genügt oft, das Verhalten rekursiver Aufrufe unabhängig von deren Implementierung zu betrachten.

8

Syntaxbeschreibung mit (E)BNF

In der Theoretischen Informatik bezeichnet der der Begriff **Sprache** eine Menge von Zeichenketten (oder **Wörtern**) über einem **Alphabet** , der zugrundeliegenden Menge von Zeichen. Diese Menge enthält alle gültigen Schreibweisen in der Sprache enthaltener Wörter und beschreibt damit die **Syntax** der betrachteten Sprache. Die sogenannte Backus-Naur-Form ist ein Formalismus zur formalen Beschreibung von Sprachen.[1]

[1] Der Formalismus BNF wurde entwickelt, um die Syntax der Programmiersprache Algol formal zu beschreiben.

Bei Beschreibungen in BNF wird zwischen sogenannten Terminalsymbolen, die in der Sprache erlaubten Zeichen entsprechen, und Nichtterminalsymbolen, die in der beschriebenen Sprache nicht vorkommen, unterschieden. Nichtterminalsymbole sind Strukturelemente, die durch Verfeinerung zu einer Folge von Terminalsymbolen **abgeleitet** werden.

Syntax arithmetischer Ausdrücke

Mit Hilfe einer BNF können wir, wie das folgende Beispiel zeigt, formal festlegen, welche Zeichenketten vollständig geklammerten arithmetischen Ausdrücken entsprechen. Wie üblich beginnen Nichtterminalsymbole mit Großbuchstaben und Terminalsymbole sind zwischen Hochkommata notiert.

```
Exp   ::= Var
        | Val
        | '(' Exp Op Exp ')'
        | Fun '(' Exps ')'

Var   ::= 'x' | 'y' | 'z'

Val   ::= Num | '-' Num

Num   ::= Digit | Digit Num

Digit ::= '0' | ... | '9'
```

```
Op    ::= '+' | '-' | '*' | '/' | '**'

Fun   ::= 'sqrt' | 'sin' | 'cos'

Exps  ::= Exp
        | Exp ',' Exps
```

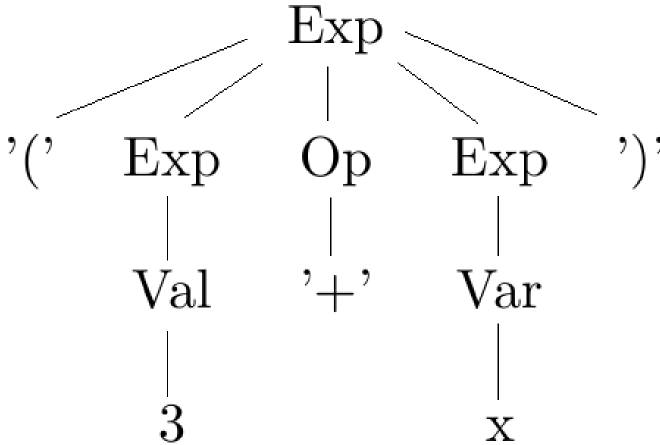
Die **Ableitungsregeln** einer BNF geben für jedes Nichtterminalsymbol hinter dem Zeichen `::=` an, wie dieses abgeleitet werden kann. Alternative Ableitungsmöglichkeiten werden dabei durch einen senkrechten Strich `|` getrennt.

Aus einer BNF kann man die in der beschriebenen Sprache enthaltenen Wörter schrittweise ableiten. Dazu beginnt man mit einem Nichtterminalsymbol und ersetzt schrittweise Nichtterminalsymbole durch eine mögliche Ableitung, bis alle Nichtterminalsymbole ersetzt sind. Zum Beispiel zeigt die folgende Ableitung, dass `(sqrt(((x**2)+1))/x)` ein Wort in der beschriebenen Sprache vollständig geklammerter arithmetische Ausdrücke ist, da wir es aus dem Nichtterminalsymbol `Exp` ableiten können.

```
Exp
(Exp Op Exp)
(Exp/Exp)
(Exp/Var)
(Exp/x)
(Fun(Exps)/x)
(sqrt(Exps)/x)
(sqrt(Exp)/x)
(sqrt((Exp Op Exp))/x)
(sqrt((Exp+Exp))/x)
(sqrt((Exp+Val))/x)
(sqrt((Exp+Num))/x)
(sqrt((Exp+Digit))/x)
(sqrt((Exp+1))/x)
(sqrt(((Exp Op Exp)+1))/x)
(sqrt(((Exp**Exp)+1))/x)
(sqrt(((Exp**Val)+1))/x)
(sqrt(((Exp**Num)+1))/x)
(sqrt(((Exp**Digit)+1))/x)
(sqrt(((Exp**2)+1))/x)
(sqrt(((Var**2)+1))/x)
(sqrt(((x**2)+1))/x)
```

Wie wir sehen, können solche Ableitungen aufwändig werden, insbesondere deshalb, weil sich von einem Schritt zum nächsten nur wenig ändert und der Rest des Wortes unverändert übernommen werden muss. Statt Ableitungen wie oben

gezeigt zu notieren, können wir sie auch mit einem sogenannten **Ableitungsbaum** darstellen. Zum Beispiel beschreibt der folgende Ableitungsbaum die Ableitung des Wortes `(3+x)` aus dem Nichtterminalsymbol `Exp`.



Die Wurzel des Ableitungsbaums ist mit dem Nichtterminalsymbol `Exp` beschriftet, aus dem das Wort `(3+x)` abgeleitet wird.

Im Allgemeinen bilden die Nichtterminalsymbole die inneren Knoten des Baums. Die Kindknoten eines inneren Knotens entsprechen der rechten Seite der Regel, die im entsprechenden Ableitungsschritt angewendet wurde. So hat die Wurzel des gezeigten Ableitungsbaums fünf Kinder, die der rechten Seite der als erstes angewendeten Regel `Exp ::= '(' Exp Op Exp ')'` entsprechen.

Die Blätter des Ableitungsbaums sind mit Terminalsymbolen beschriftet. Das abgeleitete Wort ergibt sich, indem man die Blätter des Baums (die sogenannte Front) von links nach rechts liest. Hier ergibt sich das Wort `(3+x)`.

Syntax von Palindromen

Der Formalismus BNF ist ein *universeller* Formalismus zur Beschreibung von Sprachen, also nicht nur zur Beschreibung arithmetischer Ausdrücke geeignet. Als weiteres Beispiel einer mit BNF beschriebenen Sprache betrachten wir die Spache der Palindrome.

Ein Palindrom ist ein Wort, das von vorne und von hinten gelesen gleich ist. Beispiele sind **otto**, **rentner**, oder (wenn wir Satz- und Leerzeichen sowie Groß- und Kleinschreibung vernachlässigen) **O Genie, der Herr ehre Dein Ego.** Die folgende

BNF beschreibt formal die Sprache der Palindrome über dem Alphabet `{a,...,z}`.

```
Pali ::= 'a' Pali 'a' | ... | 'z' Pali 'z'
       | 'a' | ... | 'z'
       | ''
```

Die letzte Regel erlaubt es, das Nichtterminalsymbol `Pali` zum leeren Wort, also dem Wort, das keine Zeichen enthält, abzuleiten. Dadurch wird es möglich, auch Palindrome mit gerader Anzahl Buchstaben abzuleiten.

Erweiterte BNF

Bei der formalen Spezifikation von Sprachen mit Hilfe der BNF fällt auf, dass häufig ähnliche Konstruktionen auftreten, wie zum Beispiel das optionale Vorkommen von Zeichen oder deren optionale Wiederholung. Um solche Konstruktionen einfacher notieren zu können, wurde die BNF um spezielle Konstrukte zur sogenannten EBNF erweitert.

- Das optionale Vorkommen eines Teilwortes wird durch eckige Klammern beschrieben. Zum Beispiel können wir die Regeln für das Nichtterminalsymbol `Val` mit Hilfe eckiger Klammern wie folgt vereinfachen: `Val ::= ['-'] Num`
- Die optionale Wiederholung eines Teilwortes wird durch geschweifte Klammern beschrieben. Zum Beispiel können wir die Regeln für das Nichtterminalsymbol `Exps` mit Hilfe geschweifter Klammern wie folgt vereinfachen: `Exps ::= Exp {',' Exp}`
- Schließlich können wir in EBNF den senkrechten Strich für Alternativen auch innerhalb von durch Klammerung kenntlich gemachten Gruppierungen verwenden. Zum Beispiel ließe sich das optionale Vorkommen eines Zeichens `a` statt als `['a']` auch als `('a' | '')` schreiben.

Durch die genannten Erweiterungen wird die Ausdruckstärke nicht verändert: In EBNF lassen sich genau die selben Sprachen beschreiben, die sich auch durch BNF beschreiben lassen.[2]

[2] Die Theoretische Informatik untersucht unterschiedliche Sprachklassen danach, durch welche Formalismen sie beschrieben werden können. Verschiedene Sprachklassen und zugehörige Mechanismen zu deren Beschreibung werden in der nach Noam Chomsky benannten Chomsky-Hierarchie zusammengefasst.

Syntax von Ruby-Anweisungen

Nachdem wir einen Teil von Ruby-Ausdrücken mit Hilfe von BNF beschrieben haben, wollen wir nun Anweisungen beschreiben. Dazu definieren wir eine EBNF mit einem Nichtterminal-

symbol `Stmt` unter Verwendung der vorher definierten Nichtterminalsymbole (insbesondere `Exp` für arithmetische und `BExp` für logische Ausdrücke, siehe Übung).

Die folgende Grafik veranschaulicht eine Hierarchie von Ruby-Anweisungen.

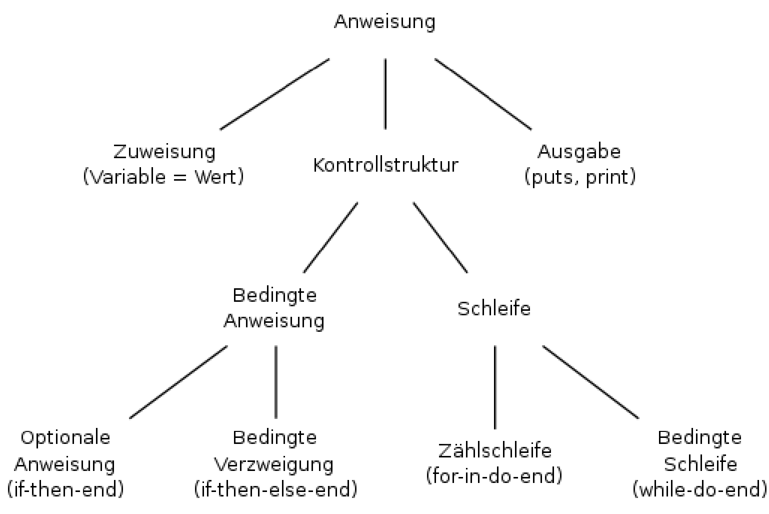


Abbildung 8.1: Anweisungs-Hierarchie

Einfache Anweisungen sind demnach Zuweisungen und Ausgabe-Anweisungen, von der wir exemplarisch die `puts`-Anweisung als mögliche Ableitung des Nichtterminals `Stmt` spezifizieren.

```
Stmt  ::= 'puts(' (Exp | BExp) ')'
```

Als Argument kann der `puts`-Anweisung ein beliebiger arithmetischer oder logischer Ausdruck übergeben werden, dessen Wert ausgegeben werden soll. Um dies zu spezifizieren, verwenden wir eine mit Klammern gruppierte Alternative der Nichtterminalsymbole `Exp` und `BExp` in der Argumentposition.

Ebenso verfahren wir bei Anweisungen, mit denen der Wert eines Ausdruck einer Variablen zugewiesen wird und erweitern entsprechend die Definition von `Stmt`.

```
Stmt  ::= ...
        | Var '=' (Exp | BExp)
```

In Ruby lassen sich mehrere Anweisungen kombinieren, indem man sie untereinander schreibt. Diese Möglichkeit formalisieren wir mit Hilfe des Nichtterminals `Stmts`

```
Stmts ::= [ Stmt { '\n' Stmt } ]
```

Hier verwenden wir eckige Klammern, um auch leere Anweisungsfolgen zu erlauben, und geschweifte, um mehrere Anweisungen durch einen Zeilenumbruch trennen zu können. Generell ignorieren wir Leerzeichen bei der Ableitung. Bei der Anwendung dieser Regel zur Ableitung von Anweisungsfolgen mit mehr als einer Anweisung müssen nach unser Definition jedoch Zeilenumbrüche vorhanden sein.

Es bleibt noch die Spezifikation von Kontrollstrukturen, also bedingten Anweisungen und Schleifen.

Bedingte Anweisungen treten in zwei Formen auf, nämlich mit und ohne Alternative hinter dem Schlüsselwort `else`. Zu ihrer Spezifikation fügen wir eine weitere Regel zur Ableitung aus dem Nichtterminal `Stmt` hinzu.

```
Stmt  ::= ...
        | 'if' BExp 'then' Stmts [ 'else' Stmts ] 'end'
```

Hier verwenden wir `BExp` für logische Ausdrücke und das eben definierte Nichtterminal `Stmts` für Anweisungsfolgen. Optionale Alternativen spezifizieren wir mit Hilfe eckiger Klammern.

Mit Schleifen können wir ähnlich verfahren. Zählschleifen definieren eine Zählvariable, einen Zahlenbereich, den diese durchläuft, und eine Anweisungsfolge, die wiederholt wird.

```
Stmt  ::= ...
        | 'for' Var 'in' Exp '..' Exp 'do' Stmts 'end'
```

Wir verwenden entsprechend das Nichtterminal `Var` für die Zählvariable, `Exp` für die Grenzen des Zahlenbereiches und `Stmts` für den Schleifenrumpf.

Schließlich fügen wir noch eine Regel zur Spezifikation bedingter Schleifen hinzu.

```
Stmt  ::= ...
        | 'while' BExp 'do' Stmts 'end'
```

Zusammengefasst ergibt sich die folgende Definition in EBNF zur Beschreibung von Ruby-Anweisungen.

```
Stmts ::= [ Stmt { '\n' Stmt } ]

Stmt  ::= 'puts(' (Exp | BExp) ')'
        | Var '=' (Exp | BExp)
        | 'if' BExp 'then' Stmts [ 'else' Stmts ] 'end'
        | 'for' Var 'in' Exp '..' Exp 'do' Stmts 'end'
        | 'while' BExp 'do' Stmts 'end'
```

9

Terme und ihre Auswertung

Zu Beginn haben wir unterschiedliche Ausdrücke in Ruby kennen gelernt. Arithmetische Ausdrücke wie `3 + Math.sqrt(x**2 + 1)`, Boole'sche Ausdrücke wie `true && (false || true)` und auch Boole'sche Ausdrücke, die als Argumente von Vergleichsoperationen arithmetische Ausdrücke enthalten wie `3*4 <= 2**3`.

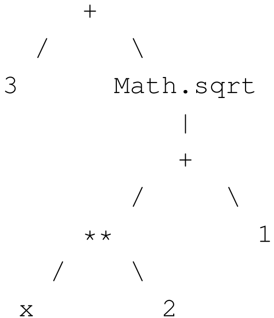
In diesem Kapitel lernen wir verschiedene Möglichkeiten kennen, Ausdrücke, im Allgemeinen auch Terme genannt, darzustellen. Basierend auf einer bestimmten Term-Darstellung lernen wir dann ein Verfahren kennen, mit dem Terme automatisch ausgewertet werden können.

Termdarstellungen

In der Informatik wird zwischen verschiedenen Darstellungsebenen von Termen unterschieden. Der selbe Term kann auf unterschiedliche Weise dargestellt werden und verschiedene Terme können zu dem selben Wert ausgewertet werden. Die Darstellung eines Terms wird als seine **Syntax** bezeichnet, der Wert zu dem er ausgewertet wird, als seine **Semantik** . Zum Beispiel sind `1 + 2` und `2 + 1` zwei verschiedene Terme mit der selben Semantik. Im folgenden werden wir sehen, dass auch ein und der selbe Term mit unterschiedlicher Syntax dargestellt werden kann.

Eine Möglichkeit, Terme auf unterschiedliche Weise darzustellen, ist es, Klammern zu schreiben, die bereits durch Präzedenzregeln implizit vorgegeben sind. Zum Beispiel sind `3 + Math.sqrt(x**2 + 1)` und `3 + Math.sqrt((x**2)+1)` der selbe Term, da Potenzierung (`**`) stärker bindet als Addition (+), die zusätzlichen Klammern an der Termstruktur also nichts ändern. Durch vollständige Klammerung kann die Struktur eines Terms ohne Hilfe von Präzedenzregeln eindeutig kenntlich gemacht werden. Eine andere Möglichkeit sind sogenannte

Termbäume, wie der folgende, der den obigen Term repräsentiert.



Hier stehen Funktionssymbole oberhalb ihrer Argumente und die Klammerung ist durch die Baumstruktur kenntlich gemacht.[1]

[1] In der Informatik wachsen Baume von oben nach unten.

In Ruby werden Funktionsnamen wie `Math.sqrt` vor ihren Argumenten notiert (**Präfix**-Notation) und zweistellige Operatoren werden zwischen ihren Argumenten notiert (**Infix**-Notation).

Wir können auch Operatoren in Präfix-Notation schreiben:

```
+(3,Math.sqrt(+(**(x,2),1)))
```

Wenn die Stelligkeit (also die Anzahl der Argumente) aller Funktions- und Operator-Symbole eideutig festgelegt ist, können wir alle Klammern weglassen, ohne dass die Termstruktur dadurch verloren geht:

```
+ 3 Math.sqrt + ** x 2 1
```

Da wir die Stelligkeiten aller Funktions- und Operator-Symbole kennen, können wir den zu dieser Darstellung gehörigen Termbaum eindeutig rekonstruieren. Wir können auch umgekehrt die Präfix-Notation aus dem Termbaum ableiten, indem wir zuerst die Wurzel des Baums notieren und dann mit den Argumenten genauso verfahren. Wir notieren also danach die Wurzel des Teilbaums für das erste Argument, dann dessen Argumente und so weiter. Wenn dieser Teilbaum abgearbeitet ist, verfahren wir entsprechend mit den weiteren Argumenten.

Analog zur Präfix-Notation wird auch die **Postfix**-Notation betrachtet. Diese kann aus dem Termbaum abgeleitet werden, indem die Wurzel jedes Teilbaums nicht vor sondern nach den zugehörigen Argumenten notiert wird. Für das obige Beispiel ergibt sich:

```
3 x 2 ** 1 + Math.sqrt +
```

Genau wie aus der Präfix-Notation kann auch aus der Postfix-Notation der zugehörige Termbaum anhand der Stelligkeiten rekonstruiert werden.

Die (wie bei der Präfix-Notation) klammerfrei eindeutige Darstellung ist nur ein Vorteil der Postfix-Notation. Der eigentlich Grund für die Relevanz der Postfix-Notation ist, dass sie sich besonders gut eignet, um Terme mit Hilfe einer sogenannten Stackmaschine auszuwerten. Bevor wir uns dem dieser Auswertung zu Grunde liegenden Mechanismus widmen, lernen wir jedoch unsere ersten Datenstrukturen kennen.

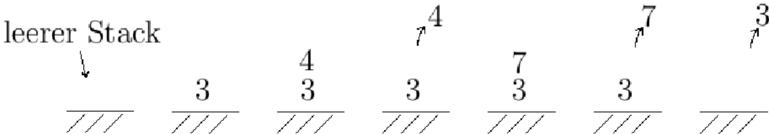
Schlangen und Keller

Datenstrukturen dienen dazu, mehrere Werte zu einem Ganzen zusammenzufassen. Zwei der einfachsten Datenstrukturen in der Informatik sind sogenannte Schlangen (englisch: queues) und Keller (auch Stapel oder englisch: stacks).

Queues arbeiten nach dem FIFO-Prinzip (**f**irst **i**n, **f**irst **o**ut). Sie stellen Operationen zum Einfügen und Entfernen von Elementen bereit, wobei, wie in einer Warteschlange, ein Element erst dann entfernt werden kann, wenn alle vor ihm eingefügten Elemente entfernt wurden.

Stacks arbeiten nach dem LIFO-Prinzip (**l**ast **i**n, **f**irst **o**ut). Elemente können auf einem Stack abgelegt und von diesem entnommen werden, wobei immer nur das zuletzt abgelegt Element entnommen werden kann. Die Operation zum Ablegen eines Elements auf dem Stack heißt traditionell `push`, die zum Entnehmen des zuletzt abgelegten Elements heißt `pop`.

Das folgende Beispiel veranschaulicht die Arbeitsweise eines Stacks anhand einiger Beispielaufrufe dieser Operationen.



Zu Beginn ist der Stack leer. Dann wird mit `push(3)` das Element `3` auf den Stack gelegt. Als nächstes wird mit `push(4)` ein weiteres Element oben auf den Stack gelegt, welches dann mit `pop()` wieder entfernt wird. Die Operation `pop` benötigt kein Argument, da immer nur das oberste Element entfernt werden kann. Im Anschluss werden noch die Operationen `push(7)`, `pop()` und noch einmal `pop()` ausgeführt, wonach der Stack wieder leer ist.

Im Folgenden werden wir Stacks auch horizontal notieren. Das obige Beispiel sähe in dieser Schreibweise so aus:

```
    |     # push(3)
  3 |     # push(4)
3 4 |     # pop()
  3 |     # push(7)
3 7 |     # pop()
  3 |     # pop()
    |
```

Der Stack ist zu Beginn und am Ende leer und neue Elemente werden rechts neben schon existierende eingefügt.

Termauswertung mit einer Stackmaschine

Stacks können verwendet werden, um beliebige Terme in Postfix-Notation automatisch auszuwerten. Bevor Stacks entdeckt wurden, war unklar, wie Terme mit unbegrenzter Schachtelungstiefe automatisch auszuwerten sind. Tatsächlich war in frühen Programmiersprachen die Schachtelungstiefe für Klammerung begrenzt. Erst mit der Entdeckung von Stacks konnten solche Begrenzungen aufgehoben werden.

Zur Auswertung eines Terms in Postfix-Notation wird dieser rechts neben einen leeren Stack geschrieben. Als Beispiel betrachten wir die Auswertung unseres Beispielausdrucks für die Variablenbelegung `x = 0`.

```
      | 3 0 2 ** 1 + Math.sqrt +
```

Ist das am weitesten links stehende Symbol wie hier eine Konstante, wird es mit der Operation `push` auf den Stack gelegt und aus der Termdarstellung entfernt:

```
      | 3 0 2 ** 1 + Math.sqrt +    # push(3)
    3 | 0 2 ** 1 + Math.sqrt +      # push(0)
  3 0 | 2 ** 1 + Math.sqrt +        # push(2)
3 0 2 | ** 1 + Math.sqrt +
```

Ist das am weitesten links stehende Symbol wie hier ein Funktions- oder Operator-Symbol, werden zuerst Elemente entsprechend der Stelligkeit des Symbols mit `pop` vom Stack entfernt und dann das Ergebnis der Anwendung der zum Symbol gehörigen (hier mathematischen) Funktion auf diese Argumente mit `push` oben auf den Stack gelegt.

```
3 0 2 | ** 1 + Math.sqrt +     # pop(); pop(); push(0**2)
  3 0 | 1 + Math.sqrt +
```

Während `**` in der Termdarstellung ein Funktionssymbol (Syntax) bezeichnet, bezeichnet es im Argument von `push` die zugehörige mathematische Funktion zur Potenzierung (Semantik). Entsprechend steht nach der Abarbeitung dieses Schrittes der Wert `0 = 0**2` oben auf dem Stack. Nun verfahren wir gemäß dieser Regeln, bis der komplette Term abgearbeitet ist und auf dem Stack nur noch ein einziger Wert steht.

```
  3 0 | 1 + Math.sqrt +        # push(1)
3 0 1 | + Math.sqrt +          # pop(); pop(); push(0+1)
  3 1 | Math.sqrt +            # pop(); push(Math.sqrt(1))
  3 1 | +                      # pop(); pop(); push(3+1)
    4 |
```

Die Auswertung des Terms `3 + Math.sqrt(0**2 + 1)` endet also mit dem Ergebnis `4`.

Tabellenkalkulation

Office-Programme zur Tabellenkalkulation bieten die Möglichkeit, den Inhalt von Tabellenfeldern mit Hilfe von Termen automatisch berechnen zu lassen. Die Rolle von Variablen spielen Feldbezeichner wie `A1`, `B7` und so weiter.

Wir können zum Beispiel die Formel `=(2-1 > 0) AND (SIN(A1) < 0.01)` in das Feld `A2` eintragen[2], die den Term aus der obigen Übungsaufgabe für den Feldbezeichner `A1` statt der Variablen `x` ausrechnet. Schreiben wir den Wert `3.14` in das Feld `A1`, so wird in das Feld `A2` automatisch der Wert `TRUE` eingetragen.

[2] Die Syntax variiert je nach verwendeter Software.

10

Objekte und ihre Identität

Bisher haben wir Zahlen und Zeichenketten als primitive Werte betrachtet und auch schon zusammengesetzte Werte in Form von Arrays kennengelernt. In Ruby werden solche Werte als sogenannte **Objekte** dargestellt, die den Zugriff auf einen internen Zustand mit sogenannten **Methoden** erlauben. Zum Beispiel ist `to_s` eine Methode von `Fixnum`-Objekten, die eine `String`-Darstellung des internen Zustands eines `Fixnum`-Objektes (also des Wertes der Zahl) zurück liefert.

Mutation

Bei Arrays haben wir auch schon gesehen, dass wir den internen Zustand ändern können. Zum Beispiel haben wir

```
factorials[i] = i * factorials[i-1]
```

geschrieben, um in einer Schleife mit der Zählvariablen `i` ein Array von Fakultäten zu erzeugen. Solche sogenannten Mutationen von Objekten können auch in Prozeduren abstrahiert werden. ein typisches Beispiel ist die Prozedur `swap!`, die zwei Elemente in einem Array vertauscht.[1]

[1] In Ruby wird mutierenden Prozeduren in der Regel ein Ausrufezeichen angehängt. Dies ist lediglich eine Namenskonvention, die kenntlich macht, dass eine Prozedur den internen Zustand eines Objektes verändern kann.

```
def swap!(a,i,j)
  x = a[i]
  a[i] = a[j]
  a[j] = x
end
```

Der Rumpf der Prozedur `swap!` enthält zwei Mutationen des in dem Parameter `a` gespeicherten Arrays.

Wir können unter Verwendung von `swap!` kompliziertere mutierende Prozeduren definieren; zum Beispiel eine, die die Reihenfolge der Elemente eines Arrays umkehrt.

```
def reverse!(a)
  for i in 0..a.size/2-1 do
    swap!(a,i,a.size-i-1)
  end
end
```

Objekt-Identität

Wir wollen in diesem Kapitel das Verhalten von Programmen mit Mutationen genauer verstehen lernen. Dazu betrachten wir zunächst das folgende Programm.

```
a = [1,2,3]
b = [1,2,3]
reverse!(a)
p(b)
```

Die Ausgabe dieses Programms ist (wie zu erwarten ist) `[1,2,3]`, da zwar die Reihenfolge der Elemente von `a` umgekehrt, dann aber der Wert von `b` ausgegeben wird, der nicht verändert wurde.

Durch eine kleine Änderung ändert sich die Ausgabe dieses Programms.

```
a = [1,2,3]
b = a
reverse!(a)
p(b)
```

In der zweiten Zeile wird jetzt der Variablen `b` der Wert von `a` zugewiesen; die anderen Zeilen bleiben unverändert. Durch diese Änderung gibt das Programm nicht mehr `[1,2,3]` aus sondern `[3,2,1]` also den Wert des umgekehrten Arrays `a`, obwohl noch immer `b` ausgegeben wird. Der Grund für dieses Verhalten ist, dass `a` und `b` als Werte *das selbe Array* haben und nicht nur wie vorher Arrays *mit den selben Elementen*.

Obwohl also im ersten Programm die `a` und `b` zugewiesenen Arrays die selben Elemente enthalten, handelt es sich bei ihnen um unterschiedliche Objekte mit unterschiedlichen Identitäten. Die Veränderung des Zustands des einen Objektes hat keinen Einfluss auf den Zustand des anderen.

Im zweiten Programm hingegen wird nur ein Array-Objekt erzeugt und als Wert den Variablen `a` und `b` zugewiesen. Dadurch ändert sich auch der Zustand des in `b` gespeicherten Arrays, sobald der Zustand des in `a` gespeicherten Arrays geändert wird, da es sich dabei um das selbe Objekt handelt.

Um diesen Effekt besser zu verstehen, können wir Objekte als Kästen zeichnen, in die wir ihren Zustand schreiben und Variablen als Referenzen auf Objekte, die auf entsprechende Kästen zeigen. Für das erste Programm ergeben sich dabei zwei Kästen mit gleichem Zustand, auf die jeweils eine Variable zeigt. Im zweiten Programm ergibt sich nur ein Kasten, auf den zwei Variablen zeigen.

Der Unterschied zwischen identischen Objekten und solchen, deren Zustände lediglich den gleichen Wert haben, kann in Ruby durch unterschiedliche Vergleichsfunktionen beobachtet werden. Der Vergleichsoperator == vergleicht die Werte von Objekten während die Methode `equal?` deren Identität vergleicht. Die folgenden Aufrufe demonstrieren diesen Unterschied.

```
irb> a = [1,2,3]
irb> b = [1,2,3]
irb> c = a
irb> a == b
=> true
irb> b == c
=> true
irb> a == c
=> true
irb> a.equal?(b)
=> false
irb> b.equal?(c)
=> false
irb> a.equal?(c)
=> true
```

Mutierende Methoden

In Ruby sind für Arrays Methoden zum Umkehren der Reihenfolge ihrer Elemente vordefiniert und zwar in mutierenden und nicht mutierenden Varianten. Die Methode `reverse` liefert ein neues Array-Objekt zurück, das die Elemente in umgekehrter Reihenfolge enthält. Die Methode `reverse!` hingegen verändert den Zustand des zugehörigen Objektes (und gibt dieses auch als Ergebnis zurück).

Die folgenden Aufrufe verdeutlichen den Unterschied zwischen den Methoden `reverse` und `reverse!`.

```
irb> a = [1,2,3]
irb> a.reverse
=> [3, 2, 1]
irb> a
=> [1, 2, 3]
```

```
irb> a.reverse!
=> [3, 2, 1]
irb> a
=> [3, 2, 1]
```

Nach dem Aufruf von `reverse` bleibt der Wert von `a` also unverändert während er sich nach dem Aufruf von `reverse!` ändert.

11

Ruby-spezifische Sprachkonstrukte

Um später nicht von uns definierte Ruby Programme verwenden zu können, lernen wir in diesem Kapitel spezifische Sprachkonstrukte kennen, die darin häufig zur Anwendung kommen und in anderen Sprachen nicht immer in dieser Form zur Verfügung stehen.

Symbole, Hash-Tabellen und benannte Parameter

In Ruby können statt Zeichenketten sogenannte **Symbole** verwendet werden. Diese haben eine eingeschränktere Schnittstelle (sie bieten zum Beispiel keine Methoden zur Konkatenation oder zur Abfrage ihrer Länge) werden aber intern effizienter dargestellt. Symbole werden mit einem vorangestellten Doppelpunkt geschrieben (`:foo`, `:bar`, `:baz`).

Da Symbole effizienter verglichen werden können als Zeichenketten, eignen sie sich besonders gut als Schlüssel in sogenannten Hash-Tabellen. Hash-Tabellen ähneln Arrays, werden aber nicht über Zahlen sondern über sogenannte Schlüssel indiziert. Sie ordnen also den Schlüsseln gewisse Werte zu. Das folgende Ruby-Programm illustriert die Verwendung von Hash-Tabellen mit Symbolen als Schlüssel.

```
table = { :foo => 41, :bar => 42, :baz => 43 }
table[:foo] = 44
puts(table[:bar])
```

Dieses Programm verändert zunächst den unter dem Schlüssel `:foo` gespeicherten Wert und gibt dann den, dem Schlüssel `:bar` zugeordneten, Wert `42` auf dem Bildschirm aus. In welcher Reihenfolge die Einträge der Hash-Tabelle notiert werden, spielt hierbei keine Rolle.

Eine interessante Besonderheit ergibt sich bei Funktionen und Prozeduren, die eine Hash-Tabelle als Parameter haben.

Beim Aufruf solcher Funktionen und Prozeduren können die geschweiften Klammern weggelassen werden, so dass es aussieht als hätten sie benannte Parameter:

```
def print_name(args)
  puts(args[:first] + " " + args[:last])
end

print_name(:last => "Huch", :first => "Frank")
```

Dieses Programm gibt die Zeichenkette `"Frank Huch"` auf dem Bildschirm aus.

Alternativ können wir den Aufruf auch wie folgt schreiben:

```
print_name(last: "Huch", first: "Frank")
```

Hier ist die Hash-Tabelle kaum noch zu erkennen, die Parameter werden aber dennoch als solche übergeben.

Blöcke

Blöcke fassen Anweisungs-Sequenzen zu einer Einheit zusammen, die als Argument an Funktionen und Prozeduren übergeben werden kann ohne sie vorher auszuführen. Funktionen und Prozeduren, die einen Block als Argument erhalten, können diesen im Rumpf (auch mehrfach) ausführen.

Zum Beispiel gibt es in Ruby eine alternative Methode Zählschleifen zu definieren, die nicht auf die Zählvariable zugreifen, indem der Schleifenrumpf als Block an die Methode `times` von `Fixnum`-Objekten übergeben wird. Das folgende Programm demonstriert drei unterschiedliche Möglichkeiten, die Zeichenkette `"hahaha"` auszugeben.

```
for i in 1..3 do
  print "ha"
end

3.times do
  print("ha")
end

3.times { print("ha") }
```

Blöcke können mit dem Schlüsselwort `do` oder in geschweiften Klammern notiert werden. Die Schreibweise mit geschweiften Klammern eignet sich besonders für Blöcke mit nur einer Anweisung.

Arrays bieten Methoden mit Blöcken als Parameter, die typische Muster von Schleifenkonstrukten abstrahieren. Die Methode `each`, zum Beispiel, führt den übergebenen Block einmal für jedes Element eines Arrays aus und übergibt dabei jeweils das entsprechende Element als Argument an den Block. Blöcke können also parametrisiert werden. Das folgende Ruby-Programm zeigt die zwei Arten Blöcke zu schreiben und eine entsprechende `for`-Schleife mit dem selben Verhalten wie die beiden Aufrufe von `each`.

```
a = [41,42,43]

for i in 0..a.size-1 do
  puts a[i]
end

a.each do |n|
  puts n
end

a.each { |n| puts n }
```

Blöcke können auch hier mit dem Schlüsselwort `do` oder in geschweiften Klammern notiert werden. Die Parameter eines Blocks stehen in jedem Fall zwischen senkrechten Strichen. Der Vorteil der Verwendung von `each` ist, dass die Grenzen der Zählvariable nicht explizit angegeben werden müssen.

Die `each`-Methode liefert als Ergebnis das Array zurück, auf dem sie aufgerufen wurde. Die `collect`-Methode hingegen liefert ein neues Array, deren Elemente sich durch die Ausführung des Blockes ergeben. Zum Beispiel liefert der Ausdruck

```
[41,42,43].collect { |n| n - 40 }
```

das Array `[1,2,3]` zurück. Es ist also möglich, Ausdrücke statt Anweisungen in Blöcken zu notieren, deren Wert von der Methode, die den Block aufruft, verwendet werden kann.

Es ist auch möglich Arrays mit Hilfe von Blöcken zu filtern. Die Array-Methode `find_all` liefert ein Array als Ergebnis, in das die Elemente übernommen werden, für die der übergebene Block `true` zurück liefert. Zum Beispiel liefert

```
[41,42,43].find_all { |n| n % 2 == 1 }
```

das Array `[41,43]` zurück, also alle ungeraden Zahlen aus dem ursprünglichen Array.

Schließlich betrachten wir noch die Methode `inject`, die alle Elemente eines Arrays zu einem Ergebnis akkumuliert, das

nicht unbedingt ein Array sein muss. Zum Beispiel können wir mit `inject` wie folgt die Elemente eines Arrays addieren:

```
[41,42,43].inject(0) { |sum,n| sum + n }
```

Der Wert dieses Ausdrucks ist `126`. An diesem Beispiel sehen wir, dass die Methode `inject` neben dem Block noch einen Startwert (hier `0`) als ersten Parameter nimmt und dass sie zwei Argumente an den Block übergibt: ein Zwischenergebnis (hier `sum` genannt) und ein Element des Arrays (hier `n`).

Blöcke sind hilfreich zur Abstraktion von Schleifen. Wir werden später auch noch andere Verwendungen von Blöcken kennenlernen.

12

Definition von Objekten

Bisher haben wir vordefinierte Objekte verwendet und uns mit den Eigenheiten mutierender Methoden vertraut gemacht. In diesem Kapitel werden wir eigene Objekte definieren. Zunächst definieren wir eigene Objekte ohne mutierende Methoden zur Darstellung rationaler Zahlen. Später definieren wir Objekte zur Darstellung von Bankkonten, deren Zustand mit Hilfe mutierender Methoden verändert werden kann.

Objekte fassen einen Zustand und darauf definierte Operationen zusammen. Der Zustand wird dabei in sogenannten **Attributen** gespeichert, die Operationen mit Hilfe sogenannter **Methoden** definiert. Objekte sind immer **Instanzen** sogenannter **Klassen**, die festlegen, welche Attribute und Methoden zu ihr gehörige Objekte haben.

Rationale Zahlen als Objekte

Als Beispiel definieren wir eine Klasse `Bruch` von Objekten, die rationale Zahlen darstellen.

```
class Bruch
    def initialize(zaehler, nenner)
        @zaehler = zaehler
        @nenner = nenner
    end
end
```

Das Schlüsselwort `class` leitet die Klassendefinition ein und ist gefolgt vom Namen der Klasse. Das Schlüsselwort `end` beendet die Klassendefinition. Innerhalb der Klasse definieren wir eine, **Konstruktor** genannte, Methode `initialize`, die bei der Erzeugung von Objekten der Klasse ausgeführt wird. Die Methode hat hier zwei Parameter `zaehler` und `nenner`, die bei der Erzeugung angegeben werden müssen, und speichert deren Werte in den **Attributvariablen** `@zaehler` und `@nenner`.

Die vorangestellten @-Zeichen kennzeichnen diese Variablen als Attribute, die den Zustand konstruierter Objekte speichern und überall innerhalb der Klassendefinition (und nicht nur in der Methode initialize) sichtbar sind.

Um Objekte der Klasse Bruch zu erzeugen, schreiben wir Bruch.new und übergeben die vom Konstruktor erwarteten Parameter für Zähler und Nenner.

Bisher haben wir Objekte meist ohne Verwendung von new erzeugt. Für Zahlen, Zeichenketten, Arrays und Hash-Tabellen bietet Ruby spezielle Syntax, die es erlaubt, Objekte kompakter zu initialisieren. Bei Zeichenketten, Arrays und Hash-Tabellen können wir allerdings auch new Verwenden, wie die folgenden Aufrufe zeigen.[1]

[1] Warum dies bei Zahlen nicht möglich ist, soll uns jetzt nicht ablenken. Es stellt sicher, dass stets identische Objekte für gleiche Zahlen verwendet werden.

```
irb> Fixnum.new(42)
NoMethodError: undefined method ‘new’ for Fixnum:Class
irb> String.new("hallo")
=> "hallo"
irb> Array.new(5,42)
=> [42, 42, 42, 42, 42]
irb> h = Hash.new("mööp")
=> {}
irb> h[:gibtsnich]
=> "mööp"
irb> Array.new([1,2,3])
=> [1, 2, 3]
```

Arrays und Hash-Tabellen bieten die Möglichkeit im Konstruktor sogenannte Default-Werte zu definieren. Von Arrays kann man eine Kopie anlegen, indem man sie bei der Konstruktion eines neuen Objektes als Parameter übergibt.

Doch nun zurück zu der selbst definierten Klasse für Brüche.

```
irb> Bruch.new(3,4)
=> #<Bruch:0x000000013b58e8 @zaehler=3, @nenner=4>
irb> Bruch.new(8,6)
=> #<Bruch:0x000000013abaa0 @zaehler=8, @nenner=6>
```

Durch Übergabe von Zähler und Nenner an die Methode new wird jeweils ein neues Objekt erzeugt, das entsprechende Werte in den Attributen @zaehler und @nenner speichert. In irb werden die erzeugten Objekte standardmäßig durch Angabe des Klassennamens, der Speicheradresse und der Attribute mit Werten angezeigt. Dieses Verhalten können wir beeinflussen, indem wir eine Methode to_s definieren, die dann automatisch verwendet wird.

Dazu fügen wir innerhalb der Klassendefinition folgendes ein.

```
    def to_s()
        return (@zaehler.to_s + "/" + @nenner.to_s)
    end
```

Wir erzeugen erneut Bruch-Objekte und beobachten, wie sie nun angezeigt werden.

```
irb> drei4tel = Bruch.new(3,4)
=> 3/4
irb> acht6tel = Bruch.new(8,6)
=> 8/6
irb> puts(drei4tel)
3/4
irb> puts(acht6tel)
8/6
```

Wie wir sehen, wandelt auch `puts` das Argument automatisch mit Hilfe der `to_s`-Methode in einen String um, wenn diese vorhanden ist. Je nach verwendeter Ruby-Version kann es nötig sein, auch die Methode `inspect` zu definieren. In einigen Versionen wird diese Methode bei der Anzeige im `irb` (und auch von der Prozedur `p`) verwendet, wahrend die Prozedur `puts` die Methode `to_s` zur Anzeige verwendet.

Es wäre schön, wenn Brüche automatisch gekürzt würden. Dazu können wir Zähler und Nenner im Konstruktor durch deren größten gemeinsamen Teiler teilen. Zur Berechnung dessen verwenden wir den Algorithmus von Euklid.[2]

[2] Der Algorithmus von Euklid ist nicht nur kürzer als unsere bisher definierten Algorithmen zur Berechnung des ggT, er kommt auch schneller zum Ergebnis.

Wir ersetzen also den Konstruktor `initialize` wie hier gezeigt und fügen die Methode `ggT` hinzu.

```
    def initialize(zaehler, nenner)
        gcd = ggT(zaehler, nenner)

        @zaehler = zaehler / gcd
        @nenner = nenner / gcd
    end

    def ggT(a,b)
        while b != 0 do
            x = b
            b = a % x
            a = x
        end

        return a
    end
```

Nun werden alle erzeugten Brüche intern gekürzt dargestellt also auch so angezeigt.

```
irb> acht6tel = Bruch.new(8,6)
=> 4/3
```

Als nächstes wollen wir eine Methode zum Multiplizieren von Brüchen definieren. Diese Methode soll das Ergebnis als neues Objekt zurück liefern und die multiplizierten Objekte nicht verändern.

In Ruby wird die Schreibweise `drei4tel * acht6tel` als Methodenaufruf `drei4tel.*(acht6tel)` interpretiert. Zur Definition der Multiplikation definieren wir also eine Methode mit dem Namen `*`. Deren Implementierung erzeugt ein neues Objekt der Klasse Bruch und muss sowohl auf die eigenen Attribute zu als auch auf diejenigen des übergebenen Argumentes zugreifen.

```
def *(other)
    return Bruch.new(@zaehler*other.zaehler
                    ,@nenner*other.nenner)
end
```

Beim Versuch diese Methode auszuführen tritt jedoch ein Fehler auf, da die Methoden `zaehler` und `nenner` für Brüche nicht definiert sind.

```
irb> drei4tel * acht6tel
NoMethodError: undefined method `zaehler' for 4/3:Bruch
```

Auf Attributvariablen fremder Objekte kann also nicht ohne weiteres zugegriffen werden. Um dies zu ermöglichen, müssen wir explizit Zugrifsmethoden definieren. In der Regel verwendet man für diese Methoden den selben Namen wie für die Attributvariablen nur ohne das `@`-Zeichen.

```
def zaehler
    return @zaehler
end

def nenner
    return @nenner
end
```

Nach Definition dieser beiden Methoden innerhalb der Klasse `Bruch` funktioniert die Multiplikationsmethode.

```
irb> drei4tel * acht6tel
=> 1/1
```

Nachdem wir eben explizit fehlende Methoden zur Verfügung stellen mussten, wäre es ebenso wünschenswert, eine zu verstecken.

```
irb> drei4tel.ggT(24,16)
=> 8
```

Wir hatten nicht beabsichtigt, Brüchen die `ggT`-Funktion als sichtbare Methode hinzuzufügen. Wir wollten diese lediglich im Konstruktor verwenden, um Brüche zu kürzen.

Standardmäßig sind also keine Attribute aber alle Methoden eines Objektes von außen zugreifbar. Attribute können wir sichtbar machen, indem wir entsprechende Methoden definieren. Aber wie machen wir Methoden unsichbar?

Wenn wir innerhalb der Definition einer Klasse `private` schreiben, sind alle Methoden, die danach definiert werden von außen unsichtbar. Entsprechend können wir später auch `public` schreiben, um folgende Methoden wieder sichtbar werden zu lassen.

Hier ist noch einmal die komplette Definition der `Bruch`-Klasse inklusive Verwendung von `private` und `public`, um die `ggT`-Methode zu verstecken.

```
class Bruch
    def initialize(zaehler, nenner)
        gcd = ggT(zaehler, nenner)

        @zaehler = zaehler / gcd
        @nenner = nenner / gcd
    end

    private

    def ggT(a,b)
        while b != 0 do
            x = b
            b = a % x
            a = x
        end

        return a
    end

    public

    def to_s()
        return (@zaehler.to_s + "/" + @nenner.to_s)
    end

    def zaehler
        return @zaehler
    end
```

```
    def nenner
        return @nenner
    end

    def *(other)
        return Bruch.new(
            self.zaehler * other.zaehler,
            self.nenner * other.nenner)
    end
end
```

Zur Definition der Multiplikation verwenden wir aus ästhetischen Gründen auch für den Zugriff auf den eigenen Zustand die definierten Zugriffsmethoden. Obwohl wir auch einfach `zaehler` und `nenner` schreiben könnten, verwenden wir dazu explizit die Referenz `self` auf das Objekt, auf dem die Methode aufgerufen wurde.

Mutierbare Objekte

Als Beispiel für eine Klasse von Objekten deren Zustand veränderbar ist, implementieren wir Bankkonten, deren Guthaben zum Beispiel durch Einzahlungen verändert werden kann.

Zunächst definieren wir einen Konstruktor zum Erzeugen von Bankkonten. Dieser initialisiert das gespeicherte Guthaben mit dem Wert Null.

```
class Konto
    def initialize
        @guthaben = 0.0
    end
end
```

Wir definieren auch wieder eine Methode zum Zugriff auf das Guthaben sowie eine Methode zur Umwandlung von Konten in eine Zeichenkette.

```
    def guthaben
        return @guthaben
    end

    def to_s
        return ("Guthaben: " + guthaben.to_s)
    end
```

In der Definition von `to_s` verwenden wir die gerade definierte Methode `guthaben` statt des Attributes `@guthaben` und verzichten dabei auf die explizite Angabe von `self`.

Nun definieren wie eine Methode `einzahlen!`, die das gespeicherte Guthaben um den übergebenen Betrag erhöht.

```
def einzahlen!(betrag)
    @guthaben = guthaben + betrag

    return self
end
```

Bei mutierenden Methoden ist es üblich, das Objekt, auf dem die Methode aufgerufen wurde, selbst zurück zu liefern. Dies ermöglicht es, mehrere Veränderungen auf einmal auszuführen, wie die folgenden Aufrufe zeigen.

```
irb> k = Konto.new
=> Guthaben: 0.0
irb> k.einzahlen!(100)
=> Guthaben: 100.0
irb> k.einzahlen!(100).einzahlen!(100)
=> Guthaben: 300.0
```

Analog zum Einzahlen können wir auch das Abheben von einem Bankkonto implementieren.

```
def abheben!(betrag)
    @guthaben = guthaben - betrag

    return self
end
```

Mit der bisherigen Implementierung können wir verschiedene Konten anlegen und diese unabhängig voneinander manipulieren. Mit Hilfe einer Überweisung können wir auch Transaktionen zwischen verschiedenen Konten implementieren. Die folgende Methode tut dies.

```
def ueberweisen!(konto, betrag)
    self.abheben!(betrag)
    konto.einzahlen!(betrag)

    return self
end
```

Hier verwenden wir wieder `self`, um deutlich zu machen von welchen Konto abgehoben und auf welches eingezahlt wird.

Die folgenden Aufrufe verdeutlichen den Effekt einer Überweisung.

```
irb> k1 = Konto.new
=> Guthaben: 0.0
irb> k1.einzahlen!(100)
=> Guthaben: 100.0
irb> k2 = Konto.new
=> Guthaben: 0.0
irb> k1.ueberweisen!(k2, 70)
=> Guthaben: 30.0
irb> k2
=> Guthaben: 70.0
```

13 Sortieren und Effizienz

In diesem Kapitel beschäftigen wir uns mit Algorithmen zum Sortieren von Daten und lernen Methoden kennen, um die Effizienz von Algorithmen systematisch zu beschreiben. Zunächst wird es um einfache Sortieralgorithmen gehen, deren Laufzeit wir beispielhaft untersuchen. Dann lernen wir Methoden kennen, die Laufzeit von Algorithmen systematisch zu beschreiben und wenden diese auf die kennen gelernten Sortieralgorithmen an. Schließlich wird es um effizientere Sortieralgorithmen und einen Vergleich ihrer Laufzeit gehen.

Einfache Sortierverfahren und ihre Laufzeit

Im Folgenden entwickeln wir Prozeduren, die ein Array als Argument erwarten und als Seiteneffekt die Elemente im gegebenen Array sortieren. Als Elemente werden wir Zahlen verwenden, die vorgestellten Sortierverfahren sind jedoch meist auch zum Sortieren komplexerer Daten geeignet (sofern diese in einer gewissen Ordnung zueinander stehen).

Selection Sort

Ein einfaches Verfahren zum Sortieren lässt sich umgangssprachlich wie folgt beschreiben.

- Vertausche das erste Element mit dem kleinsten des Arrays,
- dann das zweite mit dem kleinsten im Teil ohne das erste Element,
- dann das dritte mit dem kleinsten im Teil ohne die ersten beiden,
- und so weiter bis das ganze Array durchlaufen wurde.

Dieses Verfahren heißt Selection Sort (oder Min Sort), weil die Elemente des Arrays nacheinander mit dem Minimum getauscht werden, das aus dem Teilarray aller folgenden Elemente ausgewählt wird. Um es in Ruby zu implementieren, durchlaufen wir in einer Zählschleife alle Elemente des gegebenen Arrays und vertauschen sie mit dem Minimum im Rest-Array.

```
def min_sort!(a)
  for i in 0..a.size-1 do
    swap!(a,i,min_pos(a,i))
  end
end
```

Dass die Prozedur `min_sort!` ihr Argument verändert, kennzeichnen wir durch ein Ausrufezeichen in ihrem Namen. Die Prozedur `swap!`, die zwei Elemente eines Arrays vertauscht ist wie folgt definiert:

```
def swap!(a,i,j)
  temp = a[i]
  a[i] = a[j]
  a[j] = temp
end
```

Es fehlt noch die Definition der Funktion `min_pos`, die die Position des kleinsten Elementes eines Arrays ab einer gegebenen Position liefert.

```
def min_pos(a,from)
  pos = from                      #1
  for i in (from+1)..a.size-1 do  #2
    if a[i] < a[pos] then         #3
      pos = i                     #4
    end
  end
  return pos                      #5
end
```

Diese Funktion durchläuft das Array ab der gegebenen Position `from` und merkt sich die Position `pos` des kleinsten bisher gefunden Elementes, die sie am Ende zurück liefert.

Die folgende Programmtabelle dokumentiert die Ausführung des Aufrufs `min_pos([1,2,5,3,4],2)`.

PP	pos	i	a[i] < a[pos]	return
#1	2			
#2		3		
#3			true	

PP	pos	i	a[i] < a[pos]	return
#4	3			
#2		4		
#3			false	
#5				3

Die Korrektheit dieser Funktion können wir mit Hilfe der folgenden Beobachtungen einsehen.

1. Vor dem Eintritt in die Schleife ist `pos = from`.
2. Nach jedem Schleifendurchlauf ist `pos` die Position des kleinsten Elementes in `a` zwischen `from` und `i`.
3. Nach Ausführung der Schleife ist `pos` also die Position des kleinsten Elementes zwischen `from` und dem Ende des Arrays.

Denken wir uns `i = from` in der Situation vor Eintritt in die Schleife, dann gilt die zweite Bedingung vor, während und nach der Ausführung der Schleife und heißt deshalb *Schleifen-Invariante*.

Auch von der Korrektheit der Prozedur `min_sort!` können wir uns mit Hilfe einer Invariante überzeugen. Nach jedem Schleifendurchlauf ist nämlich das Teil-Array zwischen Position `0` und `i` sortiert. Insbesondere ist also nach Durchlauf der Schleife das gesamte Array sortiert.

Wir können uns dies anhand eines Beispiels veranschaulichen, bei dem wir nacheinander Werte des sortierten Arrays notieren, wenn dieses verändert wird. Im nächsten Schritt vertauschte Elemente sind dabei hervorgehoben. Falls nur ein Element hervorgehoben ist, wird es im nächsten Schritt mit sich selbst vertauscht.

- [**1**,2,5,3,4]
- [1,**2**,5,3,4]
- [1,2,**5**,**3**,4]
- [1,2,3,**5**,**4**]
- [1,2,3,4,5]

Die Laufzeit der Prozedur `min_sort!` untersuchen wir experimentell, indem wir sie auf Arrays unterschiedlicher Größe anwenden. Wir fangen mit einem Array der Größe 1000 an, verdoppeln dann drei mal die Arraygröße und messen die Zeit, die zum Sortieren benötigt wird.

```
count = 1000
4.times do
```

```
  print(count.to_s + ": ")
  nums = Array.new(count,42)
  start = Time.now
  min_sort!(nums)
  puts(Time.now - start)
  count = 2*count
end
```

Dieses Programm gibt neben der Eingabegröße die zum Sortieren benötigte Zeit in Sekunden aus. Die Ausgabe variiert je nach Rechner auf dem das Programm ausgeführt wird. Auf meinem Laptop ergibt sich:

```
1000: 0.057372706
2000: 0.219135235
4000: 0.875965912
8000: 3.482510442
```

Wir können beobachten, dass sich die Laufzeit bei Verdoppelung der Eingabegröße jedesmal ungefähr vervierfacht. Da die Prozedur `min_sort!` nur Zählschleifen verwendet, hängt ihre Laufzeit nur unwesentlich davon ab, welche Elemente das gegebene Array enthält. Im Falle eines bereits sortierten Arrays wird der Rumpf `pos = i` der Bedingten Anweisung in der Funktion `min_pos` niemals ausgeführt, da die Bedingung `a[i] < a[pos]` immer `false` ist. Eine Zuweisung wird in der Regel jedoch neben der Vergleichsoperation vernächlässigt, die hier unabhängig von der Eingabe immer gleich häufig ausgeführt wird.

Insertion Sort

Wir lernen nun ein Sortierverfahren kennen, das im Falle eines bereits sortierten Arrays schneller ist als Selection Sort. Intuitiv verfahren wir wie beim Aufnehmen einer Hand beim Kartenspiel: neue Elemente werden der Reihe nach in ein bereits sortiertes Teil-Array eingefügt.

Zur Implementierung in Ruby durchlaufen wir die Elemente des Arrays nacheinander in einer Zählschleife. Wie bei Selection Sort soll nach jedem Schleifendurchlauf das Teil-Array von Position `0` bis zur Zählvariable `i` sortiert sein. Diesmal erreichen wir dies, indem wir das Element an Position `i` rückwärts in den bereits sortieten Teil einfügen.

```
def insertion_sort!(a)
  for i in 0..a.size-1 do
    insert_backwards!(a,i)
  end
end
```

Die Prozedur `insert_backwards!` verwendet eine bedingte Schleife um das Element an der gegebenen Position `pos` so lange mit seinem Vorgänger zu vertauschen, wie es kleiner ist als dieser.

```
def insert_backwards!(a,pos)
  while pos > 0 && a[pos] < a[pos-1] do
    swap!(a,pos,pos-1)
    pos = pos - 1
  end
end
```

Sobald das einzufügende Element nicht mehr kleiner ist als sein Vorgänger, wird die Schleife beendet. Wir brauchen es dann nicht mehr mit den davor stehenden Elementen zu vergleichen, da diese bereits sortiert sind, das einzufügende Element also nicht kleiner sein kann.

Das folgende Beispiel illustriert die Vertauschungen, die dieser Algorithmus durchführt.

- [1,2,**5**,**3**,4]
- [1,2,3,**5**,**4**]
- [1,2,3,4,5]

Systematische Laufzeitanalyse

Bisher haben wir die Laufzeit der verwendeten Sortierverfahren experimentell untersucht und einige informelle Beobachtungen angestellt, wie sich die Laufzeit für unterschiedliche Eingaben in Abhängigkeit der Eingabegröße verhält. Im Folgenden kategorisieren wir unsere Beobachtungen und lernen eine Notation kennen, um die Laufzeit von Algorithmen abstrakt zu beschreiben.

Bei Insertion Sort haben wir beobachtet, dass die Laufzeit davon abhängt, ob Elemente bereits vorsortiert sind oder nicht. Bei bereits sortierten Arrays verdoppelte sich die Laufzeit bei Verdoppelung der Eingabegröße, bei unsortierten Array vervierfachte sie sich hingegen.

Tatsächlich ist Insertion Sort bei bereits sortierten Arrays am schnellsten und bei umgekehrt sortierten Arrays am langsamsten. Es ist deshalb hilfreich, die sogenannte **Best-Case** von der **Worst-Case Komplexität** zu unterscheiden.

Statt konkreter Laufzeiten gibt man in der Regel eine Funktion an, die das Wachstum der Laufzeit in Abhängigkeit von der Eingabegröße angibt. Im Worst-Case für Insertion Sort hat sich die Laufzeit bei Verdopplung vervierfacht, bei Vervierfachung

also versechzehnfacht und so weiter. Dies entspricht der Quadratfunktion. Man sagt deshalb: “Die Worst-Case Komplexität von Insertion Sort ist quadratisch in Abhängigkeit der Größe des sortierten Arrays.”

Alternativ sagt man auch: “Die Worst-Case Komplexität von Insertion Sort ist in $O(n^2)$, wobei n die Größe des sortierten Arrays ist.” Die hier verwendete **O-Notation** hat eine genau definierte mathematische Bedeutung, die uns hier aber nicht weiter beschäftigen soll. Sie formalisiert die oben intuitiv beschriebene Angabe der Laufzeit als Funktion der Eingabegröße, hier n genannt. Dabei haben Algorithmen der Komplexität $O(1) = O(42) = O(4711)$ die gleiche abstrahierte Laufzeit. Man spricht hier auch von konstanter Laufzeit, weil diese nicht von der Eingabegröße abhängt. Außerdem gilt zum Beispiel $O(n^2) = O(\frac{n^2-n}{2})$. Die O-Notation abstrahiert die Laufzeit also so, dass von Polynomfunktionen nur der Anteil mit dem größten Exponenten von Bedeutung ist. Intuitiv wird dadurch kenntlich gemacht, wie sich die Laufzeit für sehr große Eingaben verhält. Je größer das n, desto weniger fallen die Anteile mit kleinerem Exponenten ins Gewicht. Auch konstante Faktoren (wie $\frac{1}{2}$ im obigen Beispiel) werden vernachlässigt.

Die folgende Tabelle fasst die Best- und Worst-Case Laufzeiten der definierten Sortierverfahren zusammen.

	Best-Case (sortiert)	Worst-Case (unsortiert)
Selection Sort	$O(n^2)$	$O(n^2)$
Insertion Sort	$O(n)$	$O(n^2)$

Selection Sort hat im Best- und im Worst-Case die gleiche Komplexität, während Insertion Sort im Best-Case besser ist als im Worst-Case.

Statt die Komplexitäten experimentell zu ermitteln, können wir sie auch anhand des Programms ermitteln.

Selection Sort verwendet im wesentlichen zwei geschachtelte Schleifen. Die äußere durchläuft einmal das gegebene Array, wobei in jedem Schritt die innere Schleife vom aktuellen Element bis zum Ende läuft, um das kleinste Elemente in diesem Bereich zu finden. Wenn n die Eingabegröße ist, werden (für $n > 1$) insgesamt $(n-1) + \ldots 1 = \Sigma_{i=1}^{n-1} i = \frac{(n-1)n}{2} = \frac{n^2-n}{2}$ Vergleiche ausgeführt. Für die Worst-Case Komplexität von Insertion Sort ergibt sich auf ähnliche Weise die selbe Anzahl von Vergleichen. Im Best-Case Fall wird die innere Schleife von Insertion Sort nicht ausgeführt. In diesem Fall ergeben sich also $n-1$ Vergleiche.

Neben Best- und Worst-Case Komplexität betrachtet man manchmal auch noch **Average-Case Komplexität**, also die

durchschnittliche Laufzeit gemittelt über alle möglichen Eingaben. Wir werden im nächsten Abschnitt ein Sortierverfahren kennen lernen, dessen Average-Case Komplexität sich von der Worst-Case Komplexität unterscheidet.

Effizientere Sortierverfahren

Wir lernen nun klassische rekursive Sortierverfahren kennen. Auch die Implementierung von Insertion Sort kann, wie wir gesehen haben, mit Hilfe eines rekursiven Aufrufs implementiert werden, nach dem das letzte Element an der richtigen Stelle eingefügt wird. Der Schlüssel zur Effizienz der folgenden Sortierverfahren ist es, verschiedene Teil-Arrays mit mehreren rekursiven Aufrufen zu sortieren.

Quick Sort

Die Idee von Quick Sort ist es, eine Partitionierung genannte grobe Vorsortierung durch Anwendung rekursiver Aufrufe zu vervollständigen. Die Partitionierung stellt dabei sicher, dass sich alle Elemente, die kleiner sind als ein gegebenes, im vorderen Teil und alle größeren im hinteren Teil befinden. Anschließend werden der vordere und der hintere Teil getrennt voneinander rekursiv sortiert.

Um verschiedene Teile getrennt voneinander sortieren zu können, übergeben wir als zusätzliche Parameter die Grenzen des zu sortierenden Bereiches, die mit den Array-Grenzen initialisiert werden.

```
def quick_sort!(a)
    qsort!(a,0,a.size-1)
end
```

Die rekursive Prozedur `qsort!` implementiert das beschriebene Sortierverfahren.

```
def qsort!(a,from,to)
    if from < to then
        m = partition!(a,from,to)
        qsort!(a,from,m-1)
        qsort!(a,m+1,to)
    end
end
```

Falls der zu sortierende Bereich mehr als ein Element enthält, wird er zunächst in zwei Bereiche mit der Grenze `m` partitioniert, die danach rekursiv sortiert werden. Die Prozedur `partition!`

ist eine alte Bekannte in neuem Gewand. Wir haben früher bereits ein Programm gesehen, dass ein Array auf die beschriebene Weise partitioniert. Die folgende Prozedur verallgemeinert dieses Programm so, dass die Grenzen des zu bearbeitenden Bereiches angegeben werden können.

```
def partition!(a,from,to)
    m = from                        #1
    for i in (from+1)..to do        #2
        if a[i] < a[from] then      #3
            m = m + 1               #4
            swap!(a,i,m)            #5
        end
    end
    swap!(a,from,m)                 #6

    return m                        #7
end
```

Das Element an Position `from` dient hier als sogenanntes *Partitionselement*. Die anderen Elemente des Bereiches werden so umsortiert, dass diejenigen Elemente, die kleiner sind als das Partitionselement vor allen stehen, die größer oder gleich sind. Am Ende steht das Partitionselement an Position `m` und diese Position wird zurückgegeben.

Die folgende Programmtabelle dokumentiert die Ausführung von `partition!` für die Parameter `a = [1,2,3,6,7,4,8,5]`, `from = 3` und `to = 7`.

PP	a	m	i	a[i] < a[from]	Rückgabewert
#1	[1,2,3,6,7,4,8,5]	3			
#2			4		
#3				false	
#2			5		
#3				true	
#4		4			
#5	[1,2,3,6,4,7,8,5]				
#2			6		
#3				false	
#2			7		
#3				true	
#4		5			
#5	[1,2,3,6,4,5,8,7]				
#6	[1,2,3,5,4,6,8,7]				
#7					5

Zur Evaluation der Effizienz von Quick Sort rufen wir es mit

zufälligen Arrays unterschiedlicher Größe auf. Dabei ergeben sich auf meinem Rechner die folgenden Laufzeiten.

```
1000: 0.00467453
2000: 0.006952934
4000: 0.016561893
8000: 0.031610724
16000: 0.070631159
32000: 0.160207139
64000: 0.320690295
128000: 0.728743239
256000: 1.444987597
512000: 3.253815033
```

Wir können beobachten, dass sich die Laufzeit bei Verdoppelung der Eingabegröße meist ein wenig mehr als verdoppelt. Die Laufzeit erscheint also fast linear, aber nicht ganz.

Intuitiv können wir uns den Aufwand von Quick Sort verdeutlichen, indem wir den Aufwand für die einzelnen Aufrufe von `partition!` zusammenfassen. Der erste Aufruf durchläuft das Eingabe-Array einmal komplett um es zu partitionieren. Dann folgen zwei rekursive Aufrufe von `qsort!`, deren `partition!`-Aufrufe das Array zusammengenommen ebenfalls komplett durchlaufen. Je nach Göße der dabei sortierten Bereiche folgen wieder rekursive Aufrufe, die zusammengenommen das ganze Feld durchlaufen. Um den gesamten Aufwand abzuschätzen ist also die *Rekursionstiefe* entscheidend, denn sie entscheidet, wie oft das Eingabe-Array durchlaufen wird.

Im besten Fall wird das Feld vor jedem Rekursionsschritt in gleich große Hälften partitioniert und die Rekursionstiefe ist der Logarithmus der Größe des Eingabe-Arrays. Dabei ergibt sich also eine Laufzeit in $O(n \cdot log_2(n))$. Diese Laufzeit ergibt sich auch gemittelt über alle Eingaben also im Durchschnittsfall und erklärt damit unsere experimentellen Beobachtungen.

Im schlechtesten Fall hat die eine Hälfte der Partition die Größe 1 und die andere enthält alle weiteren Elemente. Dieser Fall tritt ein, wenn das Feld sortiert oder umgekehrt sortiert ist. In diesem Fall ist die Rekursionstiefe linear in der Eingabegröße, die Laufzeit also in $O(n^2)$.

Die folgende Tabelle fasst die Laufzeiten der bisher diskutierten Sortierverfahren zusammen.

	Best-Case	Worst-Case	Average-Case
Selection Sort	$O(n^2)$	$O(n^2)$	$O(n^2)$
Insertion Sort	$O(n)$	$O(n^2)$	$O(n^2)$
Quick Sort	$O(n \cdot log_2(n))$	$O(n^2)$	$O(n \cdot log_2(n))$

Quick Sort erreicht also gegenüber den bisherigen Verfahren eine wesentliche Verbesserung im Average-Case auf Kosten einer unwesentlichen Verschlechterung im Best-Case gegenüber Insertion Sort.

Es gibt Sortier-Verfahren, die die Laufzeit auch im Worst-Case verbessern. Im Rahmen der Übung haben Sie die Möglichkeit sich mit ihnen zu befassen.

14 Rechnerarchitektur

Computer sind Maschinen zur automatischen Verarbeitung digitaler Informationen. Auf Hardware-Ebene wird die verarbeitete Information als Bit-Folge (bit für **b**inary dig**it**), das heißt als Folge von Nullen und Einsen dargestellt. Das Verhalten eines Computers kann mit Hilfe von Schaltnetzen (also Logik- und Arithmetikgattern) sowie synchronen (also getakteten) Schaltwerken vollständig beschrieben werden. Im Folgenden geben wir einen Überblick über zentrale Ideen der Technischen Informatik, die bei der Realisierung eines Digitalrechners nach Von-Neumann-Architektur von Bedeutung sind.

Schaltnetze

Da Bits nur zwei Werte annehmen (Null oder Eins, Wahr oder Falsch, Strom an oder Strom aus), können Verknüpfungen von Bits mit Hilfe logischer Operationen realisiert werden. Auch Arithmetik ist durch Kombination logischer Operationen implementierbar, indem Zahlen im Binärsystem kodiert werden.

Logikgatter

Die Verknüfungstabellen der drei gängigsten logischen Operationen sind im Folgenden dargestellt.

a	not a
0	1
1	0

a	b	a and b
0	0	0
0	1	0
1	0	0

a	b	a and b
1	1	1

a	b	a or b
0	0	0
0	1	1
1	0	1
1	1	1

Negation (not) berechnet das jeweils entgegengesetzte Bit zur Eingabe, Das Ergebnis der Konjunktion (and) ist genau dann gesetzt, wenn beide Eingaben gesetzt sind, und das Ergebnis der Disjunktion (or) ist genau dann nicht gesetzt, wenn keine der Eingaben gesetzt ist.

Jede logische Operation kann durch geeignete Kombination von *not*, *and* und *or* realisiert werden. Elektronische Bauteile, die solche Verknüpfungen implementieren, heißen Gatter oder Schaltnetze, wobei der Begriff Gatter vornehmlich für einfache Schaltnetze verwendet wird.

Die bisher gezeigten Operationen können alle mit Hilfe der sogenannten *nand* (für **n**ot **and**) Operation implementiert werden. Alle Schaltnetze eines Computers können also allein aus NAND-Gattern gebaut werden. Die Verknüpfungstabelle der nand-Operation ist wie folgt definiert.

a	b	a nand b
0	0	1
0	1	1
1	0	1
1	1	0

Die Implementierung der anderen gezeigten Operation mit Hilfe eines NAND-Gatters beschreiben wir mit Hilfe einer beispielhaft eingeführten Hardware-Beschreibungssprache. Gatter haben Ein- und Ausgänge, die konzeptuell mit Leitungen verbunden werden können. Ein NAND-Gatter hat zwei Eingänge und einen Ausgang. Verbinden wir die Eingänge mit Leitungen a und b und den Ausgang mit einer Leitung *out*, schreiben wir dies als NAND(a,b;out). Hierbei sind in der Parameter-Liste Eingänge von Ausgängen durch ein Semikolon getrennt.

Um ein NOT-Gatter zur Negation eines Bits zu Implementieren, nutzen wir aus, dass *not a* das selbe Ergebnis liefert wie a *nand* a. Ein NOT-Gatter hat einen Ein- und einen Ausgang.

```
NOT(a;out):
  NAND(a,a;out)
```

Konjunktion können wir nun mit Hilfe eines NAND- und eines NOT-Gatters implementieren, denn *a and b = not (a nand b)*:

```
AND(a,b;out):
  NAND(a,b;c)
  NOT(c;out)
```

Für die Disjunktion nutzen wir die Identität *a or b = (not a) nand (not b)*:

```
OR(a,b;out):
  NOT(a;c)
  NOT(b;d)
  NAND(c,d;out)
```

Eine häufig verwendete Verknüpfung ist *xor* (für e**x**clusive **or**), deren Ergebnis genau dann gesetzt ist, wenn die beiden Argumente unterschiedliche Werte haben:

a	b	a xor b
0	0	0
0	1	1
1	0	1
1	1	0

Die *xor*-Verknüpfung kann wie folgt als Gatter realisiert werden:

```
XOR(a,b;out):
  NOT(a;c)
  NOT(b;d)
  AND(a,d;e)
  AND(c,b;f)
  OR(e,f;out)
```

Diese Implementierung verwendet (indirekt) neun NAND-Gatter. Eine alternative Implementierung mit nur vier NAND-Gattern sieht wie folgt aus.

```
XOR(a,b;out):
  NAND(a,b;c)
  NAND(a,c;d)
  NAND(b,c;e)
  NAND(d,e;out)
```

Aus logischer Sicht ist nur das Ein-Ausgabe-Verhalten eines Gatters interessant. Allerdings beeinflusst die Anzahl der verwendeten Bauteile die Effizienz, da höhere Signallaufzeiten langsamere Berechnungen zur Folge haben.

Weitere, für die Architektur von Digitalrechnern wichtige, Schaltnetze sind sogenannte Multiplexer, die es über einen Steuerungskanal ermöglichen zwischen verschiedenen Eingängen auszuwählen. Ein 2-zu-1 Multiplexer, wählt zum Beispiel zwischen zwei Eingangs-Bits mit Hilfe eines Steuerungs-Bits aus (setzt also je nach Wert des Steuerungs-Bits den einen oder den anderen Eingang auf den Ausgang). Multiplexer können zu größeren Multiplexern zusammengeschaltet werden. Zum Beispiel kann ein 4-zu-1 Multiplexer wie folgt aus drei 2-zu-1 Multiplexern zusammengebaut werden.

```
4MUX1(x,y,a,b,c,d;out):
  2MUX1(x,a,b;e)
  2MUX1(x,c,d;f)
  2MUX1(y,e,f;out)
```

Hierbei wird je nach Wert der Steuerungs-Bits x und y einer der Eingänge a bis d auf den Ausgang *out* geleitet. Ist Implementierung des Gatters 2MUX1 ist eine Übungsaufgabe.

In Computern werden Bits in der Regel gebündelt verarbeitet. Bündelungen aus mehreren Bits heißen *Bus* und sind in der Regel 8, 16, 32, usw. Bits breit, um sie per Multiplexer effizient Steuern zu können. Digitale Multiplexer können 2^n Eingänge verarbeiten, wenn n die Anzahl der Steuerungsbits ist.

Arithmetikgatter

Schaltnetze können nicht nur logische sondern auch arithmetische Operationen ausführen, indem Zahlen als Bitfolgen, also im Binärsystem, dargestellt werden. Die folgende Tabelle zeigt die Zahlen von eins bis zehn im Dezimal- und im Binärsystem mit drei Stellen.

Anzahl	Dezimal	Binär
#	1	001
##	2	010
###	3	011
####	4	100
#####	5	101
######	6	110
#######	7	111
########	8	Überlauf
#########	9	Überlauf
##########	10	Überlauf

Addition mit Binärzahlen folgt dem gleichen Verfahren wie Addition von Zahlen in anderen Zahlensystemen: Zahlen werden stellenweise addiert, wobei Überträge zur nächsthöheren Stelle übernommen werden. Das folgende Beispiel illustriert die Addition der Zahlen zwei und drei im Binärsystem mit drei Stellen.

```
  010
+ 011
ü 1
-----
  101
```

Das Ergebnis ist die Binärdarstellung der Zahl fünf.

Zur Implementierung binärer Addition durch ein Schaltnetz implementieren wir zunächst ein Gatter HADD (für **h**alf **add**er), das aus zwei Eingangs-Bits das Ergebnis-Bit und das Übertrags-Bit berechnet.

```
HADD(a,b;sum,carry):
  XOR(a,b;sum)
  AND(a,b;carry)
```

Ein ADD-Gatter benötigt ein zusätzliches Eingabe-Bit für den Übertrag der nächst-niedrigeren Stelle. Die Definition des ADD-Gatters ist eine Übungsaufgabe. Zur Addition von Binärzahlen mit n Stellen können dann n ADD-Gatter hintereinander geschaltet werden.

Arithmetisch-logische Einheit

Die Arithmetisch-logische Einheit (ALU für **A**rithmetic-**L**ogic **U**nit) ist das komplexeste Schaltnetz im Hauptprozessor eines Computers. Sie kombiniert Implementierungen verschiedener logischer und arithmetischer Operationen, die über Steuerungs-Bits (ähnlich wie bei einem Multiplexer) ausgewählt werden können. Verschiedene Prozessoren unterscheiden sich in Art und Anzahl durch die ALU implementierter Operationen. Hierbei werden Prozessoren mit wenigen effizienten Instruktionen (RISC für **R**educed **I**nstruction **S**et **C**omputer) von solchen mit vielen maßgeschneiderten Instruktionen (MISC für **M**ultiple **I**nstruction **S**et **C**omputer) unterschieden. Der Vorteil der RISC-Architektur ist, dass die Signalverzögerung durch die ALU geringer ist, weil diese weniger Instruktionen zur Verfügung stellen muss. Der Vorteil der MISC-Architektur ist, dass sie Instruktionen, die durch mehrere RISC-Instruktionen modelliert werden müssten, direkt in Hardware und damit effizienter implementiert.

Die Ein- und Ausgabe der ALU ist mit Registern und dem Hauptspeicher verbunden, die Steuerungs-Bits werden mit Hilfe eines speziellen Registers namens Programmzähler bestimmt. Speicher und Programmzähler werden im nächsten Abschnitt behandelt. Sie sind der Schlüssel dazu, komplexe Instruktionen auf Basis der primitiven, von der ALU bereitgestellten, Instruktionen zu implementieren und deshalb ein wichtiges Abstraktionskonzept zur Realisierung von Computern.

Synchrone Schaltwerke

Prinzipiell lässt sich jede von einem Computer ausführbare Operation durch ein Schaltnetz realisieren. Allerdings wäre es unpraktisch für jede Anwendung eigens spezielle Hardware anzufertigen. Ein großer Vorteil gängiger Computer ist ihre Vielseitigkeit. Sie erlauben eine unbegrenzte Zahl unterschiedlicher Operationen mit Hilfe von Software zu realisieren. Die begrenzte Anzahl der von der ALU bereitgestellten Operationen reicht dazu aus.

Der dabei entscheidende Mechanismus ist es, mehrere Instruktionen hintereinander auszuführen und dabei auftretende Zwischenergebnisse zu speichern. Statt Schaltnetze hintereinander zu schalten um komplexe Instruktionen auszuführen, kann dabei das Ergebnis der ersten Operation gespeichert und dann *mit dem selben Schaltnetz* weiter verarbeitet werden.

Diesem Mechanismus liegt ein Konzept zugrunde, dass wir bei Schaltnetzen bisher nicht berücksichtigt haben: das der Zeit. Instruktionen werden *zeitlich* nacheinander ausgeführt und *zu einem Zeitpunkt* gespeicherte Werte können *zu einem späteren Zeitpunkt* abgefragt werden.

Zeit wird in Computern durch ein periodisches Signal modelliert. Eine Periode des Signals entspricht dabei einem Taktzyklus des Hauptprozessors. Ein Taktzyklus muss lang genug für die Signallaufzeiten aller beteiligten Schaltnetze sein. Ist dies gegeben, brauchen wir die einzelnen Signallaufzeiten nicht mehr zu berücksichtigen, um das Verhalten eines Computers zu erklären.

Flip-Flops

Schaltnetze, die zusätzlich zu ihren logischen Eingängen auch auf das Taktsignal zugreifen, heißen *synchrone Schaltwerke*. Wie bei den Schaltnetzen gibt es auch hier ein primitives Bauteil, dass allen synchronen Schaltwerken zugrunde gelegt werden kann: das Flip-Flop. Die Implementierung eines Flip-Flop ist

aus informatischer Sicht uninteressant. Wir begnügen uns damit, sein Verhalten zu beschreiben und für die Implementierung komplexerer synchroner Schaltwerke zu nutzen.

Das Verhalten eines Flip-Flops ist einfach zu beschreiben. Es hat (neben dem Eingang für das Taktsignal) einen Eingang und einen Ausgang, wobei der Ausgang immer das Eingangssignal aus dem vorigen Taktzyklus liefert.

Speicher

Ein 1-Bit-Register ist der kleinste aller Speicherbausteine. Es hat (neben einem Eingang für das Taktsignal) zwei Eingänge *in* und *load* und einen Ausgang *out*. Wenn das *load*-Bit gesetzt ist, wird der an *in* anliegende Wert gespeichert. Der Wert von *out* ist immer der momentan gespeicherte Wert. Ist das *load*-Bit nicht gesetzt, bleibt der Wert aus dem vorigen Taktzyklus gespeichert.

Die Implementierung eines 1-Bit-Registers verwendet ein Flip-Flop und einen 2-zu-1 Multiplexer, der je nach *load*-Eingang zwischen dem Eingang und dem Ausgang auswählt.

```
Reg1(clock,load,in;out):
  2MUX1(load,out,in;a)
  FlipFlop(clock,a;out)
```

Dadurch wird bei gesetztem *load*-Bit der Eingang des Registers auf den Eingang des Flip-Flops gelegt. Ist das *load*-Bit nicht gesetzt, wird das Flip-Flop mit seinem Ausgang verbunden, wodurch der Wert aus dem vorigen Taktzyklus gespeichert bleibt.

Register der Wortgröße w können aus w 1-Bit-Registern zusammengeschaltet werden. Ein- und Ausgang werden dabei zu einem Bus der Wortgröße w, deren Zustand bei gesetztem *load*-Bit komplett im Register abgelegt wird.

Hauptspeicher kann wiederum aus mehreren Registern der Wortgröße w zusammengesetzt werden. Ein- und Ausgang behalten dabei die Größe w und werden durch einen Adressierungs-Eingang erweitert, der mit Hilfe eines De-Multiplexers bestimmt, in welchem Register die angelegte Bit-Kombination abgespeichert werden soll. De-Multiplexer sind wie umgedrehte Multiplexer, leiten also ein Eingangssignal gemäß angelegter Steuerungs-Bits auf einen von mehreren möglichen Ausgängen um. Der Ausgang des Hauptspeichers ergibt sich mit Hilfe eines Multiplexers aus dem Ausgang des addressierten Registers.

Instruktionsspeicher und Programmzähler

Das vom Computer ausgeführte Programm wird in einem speziellen Bereich des Hauptspeichers (dem sogenannten Instruktionsspeicher) abgelegt, ist also nichts weiter als eine speziell interpretierte Bitfolge. Der Einfachheit halber können wir annehmen, dass jede Maschineninstruktion in einem Register des Instruktionsspeichers abgelegt ist.

Typischerweise gibt es zwei Arten von Maschineninstruktionen, die zum Beispiel durch ihr erstes Bit voneinander unterschieden werden können.

- LOAD-Instruktionen erlauben einen vorgegebenen Wert in einem Register des Hauptprozessors abzuspeichern.
- Andere Instruktionen bestehen aus Steuerungs-Bits für die ALU und Adressierungs-Bits für die Ein- und Ausgabe der von der ALU ausgeführten Operation.

Die Ausführung des im Instruktionsspeicher enthaltenen Programms steuert der sogenannte Programmzähler, der die Adresse der als nächstes auszuführenden Instruktion enthält. Der Programmzähler ist ein Register, kann also (insbesondere durch sogenannte JUMP-Instruktionen) auf eine beliebige Adresse gesetzt werden und diese speichern. Zusätzlich verfügt er in der Regel über Eingänge *reset* und *inc*. Ist das *reset*-Bit gesetzt, wird der Zähler auf Null zurückgesetzt. Ein angelegtes *inc*-Bit hat zur Folge, dass der Zähler erhöht wird, also auf die nächste Instruktion im Instruktionsspeicher zeigt.

Hauptprozessor und Von-Neumann-Architektur

Ein Computer nach Von-Neumann-Architektur besteht im Wesentlichen aus einem Hauptprozessor und einem Hauptspeicher, die über ein Bus-System verbunden sind.

Der Hauptprozessor besteht aus der ALU, aus Registern (auf die schneller zugegriffen werden kann, als auf den Hauptspeicher) sowie aus einer Steuerungseinheit bestehend aus Programmzähler und einem Schaltnetz, dass die ausgeführte Maschineninstruktion mit Hilfe der ALU verarbeitet und dabei Ein- und Ausgabe der ALU geeignet adressiert. In der Regel wird nach einer Instruktion der Programmzähler erhöht, um die nächste Instruktion auszuführen. Bei Sprungbefehlen wird er stattdessen auf die in der Instruktion angegebene Sprungadresse gesetzt.

Alle Komponenten eines Computers können letztendlich auf NAND-Gatter und Flip-Flops zurückgeführt werden. NAND-Gatter zu komplexen Schaltnetzen zusammenzuschalten ist

ein wesentliches Abstraktionsmittel, um Operationen auf Binärdaten in Hardware zu realisieren. Schaltnetze mit Flip-Flops zu synchronen Schaltwerken zu kombinieren ist das andere wesentliche Abstraktionsmittel für den Bau von Computern, denn es ermöglicht, Ergebnisse zu speichern und komplexe Instruktionen durch Hintereinader-Ausführung einfacherer Instruktionen zu implementieren.

Memory Mapped I/O

Das bisher vorgestellte Rechner-Modell bietet scheinbar keine Möglichkeit, Daten in den Computer einzugeben oder von diesem ausgeben zu lassen. Für eine informationsverarbeitende Maschine, deren einzige Aufgabe es ist, Eingabe-Information in Ausgabe-Information zu transformieren, erscheint das als ein nicht unerheblicher Nachteil.

Glücklicherweise brauchen wir die bisher vorgestellte Architektur konzeptuell nicht zu erweitern, um Ein- und Ausgabe von Daten zu ermöglichen. Durch sogenanntes *Memory-Mapped I/O* (I/O für **I**nput/**O**utput) kann der Computer auf Ein- und Ausgabegeräte zugreifen, wie auf den Hauptspeicher und so Daten einlesen oder ausgeben. Dabei wird einem angeschlossenen Gerät ein festgelegter Speicherbereich zugewiesen, der zu jedem Zeitpunkt den aktuellen Zustand des Geräts reflektiert.

Zum Beispiel kann einer Tastatur ein Register des Hauptspeichers zugeordnet werden, in dem zu jedem Zeitpunkt eine binäre Kodierung der gerade gedrückten Taste abgelegt wird. Der Computer kann dann durch Lesen dieses Registers Tastatureingaben verarbeiten.

Zur Ausgabe kann einem Bildschirm ein festgelegter Speicherbereich (zum Beispiel mit einem Register pro Bildpunkt) zugeordnet werden. Der Computer kann dann durch Schreiben in diesen Speicherbereich Ausgaben auf dem Bildschirm erzeugen.

Assembler

Algorithmen, die in Programmiersprachen formuliert sind, müssen in Maschineninstruktionen übersetzt werden, um auf einem Computer ausgeführt zu werden. Diese Aufgabe wird in der Regel von einem anderen (Compiler genannten) Programm ausgeführt. Manche Programmiersprachen (zum Beispiel C) erlauben es, sogenannte Assembler-Sprache in Programme einzubetten, um (zum Beispiel aus Effizienz-Gründen) Einfluss auf die generierten Maschineninstruktionen zu nehmen.

Als Assembler-Sprache wird eine aus den Maschineninstruktionen eines Computers abgeleitete Programmiersprache bezeichnet. Jedes Computer-Modell hat seine eigene Assembler-Sprache, die die zugrunde liegenden Maschineninstruktionen widerspiegelt. Assembler-Sprache erlaubt eine textuelle Eingabe von Maschineninstruktionen, wobei symbolische Namen für Speicheradressen benutzt werden können. Der Compiler, der Assembler-Sprache in Maschineninstruktionen übersetzt, heißt Assembler.

Als ein Beispiel für ein in (imaginärer) Assembler-Sprache geschriebenes Programm betrachten wir das folgende Programm, das die Zahlen von 1 bis 100 addiert.

```
    i = 1
    sum = 0
LOOP:
    if i = 101 goto END
    sum = sum + i
    i = i + 1
    goto LOOP
END:
    goto END
```

Die Symbole `i` und `sum` werden vom Assembler in Adressen für den Hauptspeicher (oder im Hauptprozessor enthaltene Register) übersetzt. Welche Adressen dafür verwendet werden, ist für das Verhalten des Programms irrelevant, solange sie eindeutig sind. Die Symbole `LOOP` und `END`, die in Sprungbefehlen verwendet werden, werden vom Assembler in die Adresse des Instruktionsspeichers übersetzt, in die die nach ihnen deklarierte Instruktion geschrieben wird.

Quellen und Lesetipps

- From NAND to Tetris: Building a Modern Computer From First Principles
- John von Neumann: First Draft of a Report on the EDVAC
- Wie unwichtig es ist, ob U-Boote schwimmen können (niederländisch oder englisch)

15 Digitale Bildverarbeitung

Dieses Kapitel gibt einen kleinen Einblick in die Digitale Bildverarbeitung anhand des Konzepts des Histogramms. Histogramme spielen eine zentrale Rolle für das Verständnis der digitalen Bildverarbeitung. Sie sind auch aus informatischer Sicht interessant, da sie digitale Bilder auf eine Weise zusammenfassen, die es erlaubt, Bildeigenschaften wie Helligkeit und Kontrast auf interessante Weise zu analysieren und zu manipulieren.

Im Folgenden werden wir Rastergrafiken, also solche, die als Pixelmatrix gespeichert sind, mit Hilfe gängiger Bildverarbeitungssoftware analysieren und manipulieren. Dabei werden wir Histogramme von Bildern mit unterschiedlicher Helligkeit und unterschiedlichem Kontrast betrachten und vergleichen. Später lernen wir Werkzeuge kennen, die es erlauben, diese Eigenschaften zu manipulieren. Schließlich programmieren wir Ruby-Funktionen zur Berechnung von Histogrammen sowie zur Manipulation von digitalen Bildern.

Rastergrafiken und Histogramme

Die Pixel einer Rastergrafik haben einen Farbwert, der in der Regel als Kombination aus Rot, Grün und Blau dargestellt wird. Ein Histogramm speichert zu jeder möglichen *Intensität* eines Farbwertes, wie viele Pixel mit dieser Intensität im Bild vorkommen. Unterschiedliche Histogramm-Typen unterscheiden sich dadurch, welcher Intensitätsbegriff zu Grunde gelegt wird. Als Intensität kann zum Beispiel die Gesamt-Helligkeit eines Pixels definiert werden, die Helligkeit eines einzelnen Kanals (Rot, Grün oder Blau) oder auch eine Kombination daraus.

Im folgenden betrachten wir Graustufen-Bilder, da mit ihnen bereits die grundlegenden Eigenschaften von Histogrammen demonstriert werden können.

Das Bildverarbeitungsprogramm GIMP bietet über den Menüpunkt `Tools > Color Tools > Desaturate` die Möglichkeit,

Bilder in Graustufen zu konvertieren. Dazu stehen die Optionen `Lightness`, `Luminosity` und `Average` zur Verfügung, die sich darin unterscheiden mit welcher Gewichtung die Farbinformation der einzelnen Farbkanäle zur Gesamthelligkeit kombiniert wird. Wir verwenden die Option `Luminosity`, die durch entsprechende Gewichtung berücksichtigt, dass der Grün-Anteil eines Pixels seine Helligkeit für das menschliche Auge stärker beeinflusst als Rot und Blau. Die Option `Average` gewichtet alle drei Farbkanäle gleich, während `Lightness` eine subjektive Helligkeit anhand eines komplizierteren Zusammenhangs berechnet als `Luminosity`.

Der Menüpunkt `Tools > Color Tools > Curves` öffnet einen Dialog, in dem das Histogramm eines Bildes angezeigt wird. Als erstes Beispiel betrachten wir ein dunkles Bild, das wir vorher in Graustufen umgewandelt haben.

Abbildung 15.1: Bild mit überwiegend dunklen Pixeln

Das Histogramm dieses Bildes zeigt für jeden Grauwert zwischen 0 und 255, wie viele Pixel mit diesem Grauwert im Bild vorhanden sind. Auf der x-Achse sind dazu die Grauwerte aufgetragen und auf der y-Achse die entsprechende Anzahl von Pixeln.

Wir können erkennen, dass die meisten Pixel niedrige (also dunkle) Grauwerte haben, denn diesen ist eine deutlich größere Anzahl zugeordnet als den hohen (also hellen) Grauwerten.

Am linken Rand des Histogramms ist außerdem eine Häufung zu erkennen, die andeutet, dass das Bild leicht unterbelichtet ist, also dunkle Pixel, die in der Originalszene eigentlich unterschiedliche Intensitäten hatten, alle mit dem niedrigsten Grauwert dargestellt sind. Ebenso kann es in hellen Bildern auftreten, dass viele unterschiedlich helle Stellen mit dem niedrigsten Grauwert dargestellt sind. In diesem Fall wäre das Bild überbelichtet. Spitzen an den Rändern des Histogramms sind im Fall von Fotos also ein Indikator für eine unpassende Belichtungszeit bei der Aufnahme.

Als zweites Beispiel betrachten wir nun ein helles Bild und

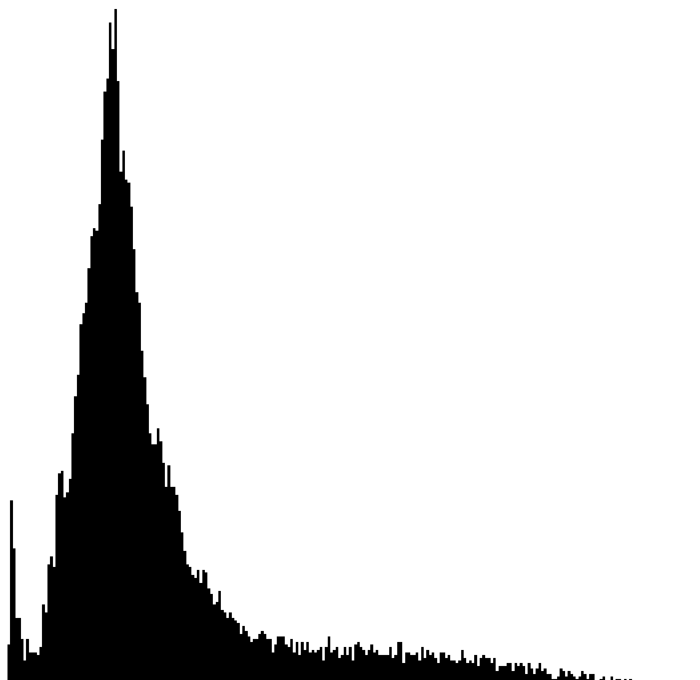


Abbildung 15.2: Histogramm eines dunklen Bildes

sein Histogramm.

Abbildung 15.3: Bild mit überwiegend hellen Pixeln

Hier sind vor allem hohen (also hellen) Grauwerten eine große Anzahl von Pixeln zugeordnet.

Die Helligkeit eines Bildes lässt sich am Histogramm also daran erkennen, ob der Grauwert-Bereich mit hohen Pixelzahlen eher links oder eher rechts im Histogramm liegt.

Als nächstes vergleichen wir die Histogramme von Bildern mit unterschiedlichem Kontrast. Dazu betrachten wir zunächst ein Bild mit hohem Kontrast, also eines in dem Pixel mit stark unterschiedlichen Graustufen häufig vorkommen.

Das Histogramm dieses Bildes ist mittig ausgerichtet, das Bild hat also eine mittlere Helligkeit.

Um zu erkennen, wie wir anhand des Histogramms auf den

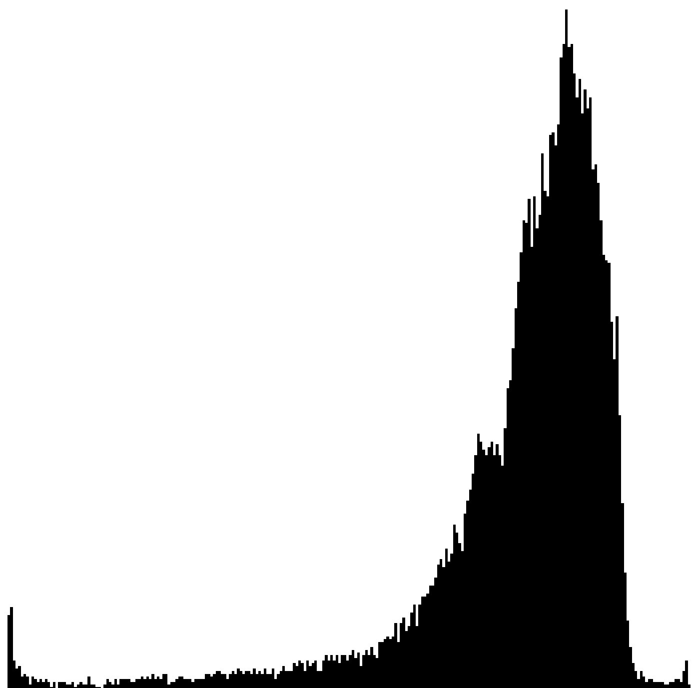


Abbildung 15.4: Histogramm eines hellen Bildes

Abbildung 15.5: Bild mit hohem Kontrast

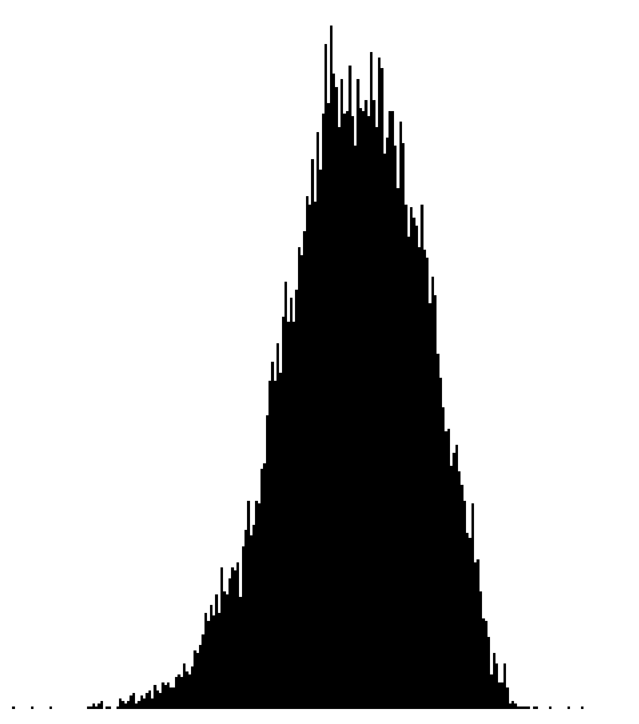


Abbildung 15.6: Histogramm eines Bildes mit hohem Kontrast

Kontrast schließen können, vergleichen wir es mit dem Histogramm eines Bildes mit niedrigerem Kontrast.

Abbildung 15.7: Bild mit niedrigem Kontrast

In diesem Bild kommen weniger unterschiedliche Pixel häufig vor. Im Histogramm ist der Grauwertbereich mit hoher Pixelzahl wieder mittig ausgerichtet, nun allerdings schmaler als im Bild mit hohem Kontrast.

Abbildung 15.8: Histogramm eines Bildes mit niedrigem Kontrast

Den Kontrast eines Bildes erkennen wir an seinem Histogramm also daran, ob sich die Grauwerte mit hoher Pixelzahl in einem schmalen Bereich konzentrieren oder weit über das Histogramm verteilt sind.

Der Dialog `Tools > Color Tools > Curves` zeigt nicht nur ein Histogramm an, sondern erlaubt es auch, das Bild mit Hilfe sogenannter Kurven zu manipulieren. Diese Kurven bilden Grauwerte auf neue Grauwerte ab. Obwohl sie im selben Koordinatensystem angezeigt werden, wie das Histogramm, sind sie keine Funktionen von Grauwerten in Pixelzahlen sondern von Grauwerten in Grauwerte. Die y-Achse ist für die Kurven also wie die x-Achse von 0 bis 255 beschriftet.

Die einfachsten Kurven sind Geraden und wir untersuchen im folgenden, wie wir mit Geraden die Helligkeit und den Kontrast von Bildern manipulieren können.

Die einfachste Möglichkeit, um die Helligkeit eines Bildes zu beeinflussen, ist, zu jedem Grauwert eine Konstante zu addieren (oder zu subtrahieren). Dies entspricht einer Verschiebung der Einheitsgeraden nach oben (oder unten), wobei sie am Rand abgeschnitten wird, um im Zielbereich der Farbwerte zu bleiben. Diese Methode, ein Bild heller zu machen, führt also bei bereits hellen Bildern zu einer künstlichen Überbelichtung. Analog führt das Verdunkeln eines bereits dunklen Bildes zu künstlicher Unterbelichtung.

Das Histogramm wird mit dieser Methode nach rechts (oder links) verschoben, da die Pixelanzahlen gleich bleiben und nur anderen Grauwerten zugeordnet werden. Dabei kann der Grauwert-Bereich mit hohen Pixelanzahlen gegen den rechten (oder linken) Rand des Histogramms gedrückt werden, wodurch die künstliche Über- (oder Unter-)belichtung sichtbar wird.

Um diesen Effekt zu vermeiden, kann man beim Aufhellen eines Bildes dunkle Pixel stärker verändern als helle und beim Verdunkeln helle stärker als dunkle. Diese Methode aum Aufhellen kann durch eine Gerade umgesetzt werden, die durch den Punkt `(255,255)` und ansonsten oberhalt der Identitätsgeraden verläuft. Analog erreicht man Verdunklung mit einer Geraden durch den Nullpunkt, die unterhalb der Identitätsgeraden verläuft.

Diese Methode verringert den Kontrast eines Bildes, da beim Aufhellen dunkle Grauwerte komplett entfernt werden. Ebenso kommen nach dem Verdunkeln ganz helle Grauwerte im Bild nicht mehr vor. Um diesen Effekt zu vermeiden, muss man Kurven verwenden, die sowohl durch den Nullpunkt als auch durch `(255,255)` aber ansonsten oberhalb (oder unterhalb) der Identitätsgeraden verlaufen.

Um den Kontrast eines Bildes zu verändern brauchen wir Kurven, die sowohl oberhalb als auch unterhalb der Identitätsgeraden verlaufen. Zum Beispiel erhöht sich der Kontrast, indem dunkle Pixel dunkler und helle heller gemacht werden. Der Kontrast verringert sich hingegen, wenn dunkle Pixel heller und helle dunkler werden. Dies erreichen wir zum Beispiel durch eine Gerade durch den Punkt `(128,128)` mit gegenüber der Identitätsgeraden erhöhter (bzw. erniedrigter) Steigung.

Im Histogramm äußert sich eine Erhöhung des Kontrasts dadurch, dass der Grauwert-Bereich mit hohen Pixelzahlen auseinandergezogen wird. Dabei kann er gegen den rechten und/oder linken Rand des Histogramms gedrückt werden, was zu künstlicher Über- und/oder Unterbelichtung führt. Dieser Effekt lässt sich vermeiden, indem Kurven gewählt werden, die

durch die Eckpunkte gehen und ansonsten sowohl unter- als auch oberhalb der Identitätsgeraden verlaufen.

Bildverarbeitung in Ruby

Wir wollen nun Ruby-Funktionen schreiben, mit denen Histogramme berechnet und Bilder manipuliert werden können. Dazu verwenden wir die Bibliothek [oily_png], die einen einfachen Zugriff auf PNG-Dateien implementiert.

Diese Bibliothek kann wie folgt installiert werden.[1]

```
# gem install oily_png
```

Nach der Installation können wir die Bibliothek mit

```
require 'oily_png'
```

in eigene Ruby-Programme einbinden. Danach haben wir Klassen `ChunkyPNG::Image` und `ChunkyPNG::Color` zum Zugriff auf Bilder und Farbwerte zur Verfügung. Der Prefix `ChunkyPNG::` kann weggelassen werden, wenn wir nach der `require`-Anweisung auch die folgende notieren.

```
include ChunkyPNG
```

Objekte der Klasse `Image` haben Methoden `width` und `height` zum Zugriff auf ihre Größe. Außerdem ist es möglich mit

```
color = image[x,y]
```

auf den Farbwert des Pixels an Position `(x,y)` zuzugreifen und diesen mit

```
image[x,y] = color
```

zu verändern, wenn `image` ein `Image`-Objekt ist. Die Farbwerte sind dabei keine Objekte der Klasse `Color` sondern Zahlen, die wir mit Hilfe von `Color` verarbeiten können. Dazu können wir zum Beispiel

```
white = Color.rgb(255,255,255)
```

schreiben, um einen weißen Farbwert zu erzeugen. Ist `color` ein Farbwert, so speichern die folgenden Zuweisungen den Rot-, Grün- bzw. Blau-Wert (zwischen 0 und 255) in einer entsprechenden Variablen.

[1] Diese Bibliothek ist eine effizientere Version der Bibliothek `chunky_png`, die sich genauso verwenden lässt. Je nach Installationsort kann es nötig sein, das Installations-Kommando mit Administratorrechten auszuführen.

```
red = Color.r(color)
green = Color.g(color)
blue = Color.b(color)
```

Objekte der Klasse `Image` können wir erzeugen, indem wir sie aus einer Datei einlesen.

```
image = Image.from_file("filename.png")
```

Alternativ können wir ein Bild durch Angabe seiner Größe und eines Farbwertes erzeugen, der für alle Pixel verwendet wird.

```
image = Image.new(width, height, Color.rgb(255,255,255))
```

Schließlich können wir Bilder auch abspeichern, indem wir die Methode `save` verwenden.

```
image.save("filename.png")
```

Als erstes Beispiel für Bildverarbeitung in Ruby definieren eine destruktive Prozedur zur Umwandlung eines Bildes in Graustufen.

```
def desaturate!(image)
    for y in 0..image.height-1 do
        for x in 0..image.width-1 do
            gray = average(image[x,y])
            image[x,y] = Color.rgb(gray, gray, gray)
        end
    end
end
```

Sie durchläuft alle Pixel des Bildes, berechnet den Grauwert mit Hilfe einer noch zu definierenden Funktion `average` und überschreibt dann den aktuellen Pixel mit seinem Grauwert. Die Funktion `average` berechnet zunächst die Rot-, Grün- und Blauwerte des übergebenen Farbwerts und gibt dann deren Mittelwert zurück. Zur Berechnung des Mittelwerts verwenden wir Gleitkommazahlen, um das Ergebnis korrekt zu runden.

```
def average(color)
    r = Color.r(color)
    g = Color.g(color)
    b = Color.b(color)

    return ((r+g+b)/3.0).round
end
```

Die folgende Prozedur liest ein Bild aus einer Datei ein, wandelt es in Graustufen um und speichert es mit einem anderen Namen ab.

```
def save_desaturated(base_name)
    image = Image.from_file(base_name + ".png")
    desaturate!(image)
    image.save(base_name + "_gray.png")
end
```

Dazu wird der Teil des Dateinamens vor der Dateiendung `.png` übergeben. Wenn die eingelesene Datei den Namen `filename.png` hat, muss also `"filename"` übergeben werden. Das Graustufenbild wird dann in einer Datei mit dem Namen `filename_gray.png` abgespeichert.

Als nächstes definieren eine Funktion zur Berechnung eines Histogramms der Grauwerte eines Bildes.

```
def gray_histogram(image)
    histogram = Array.new(256, 0)

    for y in 0..image.height-1 do
        for x in 0..image.width-1 do
            gray = average(image[x,y])
            histogram[gray] = histogram[gray] + 1
        end
    end

    return histogram
end
```

Dazu erzeugen wir ein Array aus 256 Zahlen, einer für jeden Grauwert. Dieses Array füllen wir dann, indem wir alle Pixel durchlaufen, den Grauwert jedes Pixels berechnen und jedesmal die entsprechende Anzahl erhöhen.

Aus dem Histogramm lassen sich, ohne Kenntnis des Bildes, interessante Eigenschaften berechnen. Als Beispiel definieren wir eine Funktion, die die mittlere Helligkeit eines Bildes nur anhand seines Histogramms berechnet. Dazu berechnen wir gleichzeitig die Anzahl der Pixel und die Summe der Grauwerte aller Pixel. Letztere berechnen wir, indem wir jeden Grauwert mit der Anzahl der Pixel mit diesem Grauwert multiplizieren und die Ergebnisse addieren.

```
def mean_brightness(histogram)
    total_gray = 0
    pixel_count = 0
```

```
    for gray in 0..255 do
        count = histogram[gray]
        total_gray = total_gray + gray*count
        pixel_count = pixel_count + count
    end

    return (1.0 * total_gray / pixel_count).round
end
```

Für das dunkle Bild vom Anfang dieses Abschnitts ergibt sich eine mittlere Helligkeit von 63, für das helle Bild eine von 189.

Zur Manipulation von Bildern in Graustufen können wir, wie in GIMP gesehen, Abbildungen von Grauwerten in Grauwerte verwenden. Diese stellen wir in Ruby als Arrays der Größe 256 dar, deren Einträge Zahlen zwischen 0 und 255 sind. So dargestellte Abbildungen können wir mit der folgenden Funktion auf Bilder anwenden.

```
def change_pixels!(image, gray_map)
    for y in 0..image.height-1 do
        for x in 0..image.width-1 do
            gray = gray_map[average(image[x,y])]
            image[x,y] = Color.rgb(gray, gray, gray)
        end
    end
end
```

Diese Funktion durchläuft alle Pixel und berechnet den neuen Grauwert anhand des alten und der übergebenen Abbildung.

Als Beispiel für eine Abbildung von Grauwerten, berechnen wir eine zum Aufhellen (oder Verdunkeln) eines Bildes durch Addition (oder Subtraktion) einer Konstante.

```
def brightness_adjustment(diff)
    gray_map = Array.new(256)

    for gray in 0..255 do
        new_gray = gray + diff

        new_gray = [0, new_gray].max
        new_gray = [new_gray, 255].min

        gray_map[gray] = new_gray
    end

    return gray_map
end
```

Diese Funktion erzeugt eine Abbildung als Array und weist dann jedem Grauwert den Grauwert zu, auf den er abgebildet werden soll. Dazu wird die übergebene Konstante auf den aktuellen Grauwert addiert. Subtraktionen werden durch negative Parameter erreicht. Bevor ein Grauwert gespeichert wird, wird er durch Vergleich mit 0 und 255 auf den Zahlenbereich für Grauwerte eingeschränkt.

Die Prozedur `save_with_new_brightness` ändert die mittlere Helligkeit eines mit dem gegebeben Namen gespeicherten Bildes auf den übergebenen Wert und speichert es unter einem neuen Namen ab.

```
def save_with_new_brightness(base_name, new_mean)
    image = Image.from_file(base_name + ".png")
    gray_hist = gray_histogram(image)
    old_mean = mean_brightness(gray_hist)
    gray_map = brightness_adjustment(new_mean - old_mean)
    change_pixels!(image, gray_map)
    image.save(base_name + "_luma" + new_mean.to_s + ".png")
end
```

Die Prozedur berechnet zunächst ein Histogram und daraus dann die mittlere Helligkeit des Bildes. Aus der Differenz der aktuellen und der übergebenen Helligkeit wird eine Abbildung von Grauwerten berechnet, die die Helligkeit entsprechend anpasst. Diese wird schließlich auf das eingelesene Bild angewendet, bevor es unter einem neuen Namen gespeichert wird.

Auf ähnliche Weise können wir beliebige Abbildungen von Grauwerten berechnen und zur Manipulation von Bildern auf diese anwenden. Zum Beispiel könnten wir die Helligkeit mit Hilfe von Geraden durch einen Eckpunkt verändern oder sogar kurvige Kurven als `gray_map` darstellen.

Quellen und Lesetipps

Die Bilder sind einem Histogramm Tutorial der Online Community *Cambridge in Colour* entnommen.

16

Reguläre Ausdrücke

Ein häufig wiederkehrendes Problem besteht darin, Zeichenketten zu durchsuchen oder zu manipulieren. Wir haben bereits früher Programme geschrieben, die nach einer Teilzeichenkette suchen oder eine solche ersetzen. Da dieses Problem so häufig auftritt, wird dazu in der Regel ein Mechanismus verwendet, der ohne explizite Programmierung auskommt. Dieser erlaubt es auch, statt konkrete Teilzeichenketten zu suchen oder zu ersetzen, ganze Klassen von Teilzeichenketten anzugeben.

Diese Klasse sind Mengen von Wörtern, also Sprachen, die wir mit Hilfe von (E)BNF beschreiben könnten. Reguläre Ausdrücke sind eine einfachere Methode, Sprachen (also Mengen von Wörtern) zu beschreiben. Sie erlauben effizientere Methoden zur Implementierung von Such- und Ersetzungs-Algorithmen als (E)BNF, sind jedoch auch nicht so ausdurcksstark.[1] Es gibt also Sprachen, die mit (E)BNF nicht aber mit Regulären Ausdrücken beschrieben werden können. Die früher beschriebene Sprache für arithmetische Ausdrücke ist zum Beispiel so eine Sprache. Reguläre Ausdrücke sind jedoch ausdrucksstark genug für viele relevante Problemstellungen und werden wegen ihrer vergleichsweisen Einfachheit oft bevorzugt.

[1] Es gibt Erweiterungen von Regulären Ausdrücken, die ihre Ausdrucksstärke erhöhen, die wir aber hier nicht thematisieren.

Reguläre Ausdrücke werden in Ruby zwischen Schrägstrichen notiert.[2] Im einfachsten Fall bestehen Sie einfach aus einer Zeichenkette wie zum Beispiel `/elf/`. Der Operator `=~` (in Wahrheit eine Methode auf Zeichenketten) wird verwendet, um mit Hilfe von Regulären Ausdrücken zu suchen.

[2] Reguläre Ausdrücke sind Werte, können also wie Zeichenketten oder Zahlen in Variablen und Datenstrukturen gespeichert werden.

```
irb> "Spielfigur" =~ /elf/
=> 3
irb> "Spielfigur" =~ /elfen/
=> nil
```

Das Ergebnis von `=~` ist der kleinste Index, an dem ein Wort der beschriebenen Sprache gefunden wurde, oder `nil`, falls kein solcher Index existiert.

Bestimmte Zeichen in regulären Ausdrücken werden nicht als solche interpretiert sondern haben eine besondere Bedeutung. Zum Beispiel steht der Punkt für ein beliebiges Zeichen und nicht für einen Punkt. Auch ist es möglich, explizit Gruppen von Zeichen zu definieren und auch Gruppen, die alle nicht genannten Zeichen beschreiben.

```
irb> "Spielfigur" =~ /e.f/
=> 3
irb> "Spielfigur" =~ /e[a-z]f/
=> 3
irb> "Spielfigur" =~ /e[^A-Z]f/
=> 3
```

Für bestimmte Gruppen sind Sonderzeichen vordefiniert:

- `\d` steht für eine beliebige Ziffer, also `[0-9]`,
- `\w` steht für ein Wortzeichen, also `[a-zA-Z0-9]` und
- `\s` steht für ein beliebiges Leerzeichen (inklusive Tabulatoren und Zeilenumbrüchen).

Die Ausführung von =~ modifiziert als Seiteneffekt bestimmte globale Variablen, in denen Teile der verarbeiteten Zeichenkette gespeichert werden.

```
irb> "Spielfigur" =~ /e.f/
=> 3
irb> $`
=> "Spi"
irb> $&
=> "elf"
irb> $'
=> "igur"
```

Das Fragezeichen kennzeichnet ein optionales Zeichen.

```
irb> "Spielfigur" =~ /elfi?/
=> 3
irb> $&
\=> "elfi"
irb> "Spielfigur" =~ /elfe?/
=> 3
irb> $&
=> "elf"
```

Durch Klammerung kann es auch auf mehrere Zeichen angewendet werden.

```
irb> "Spielfigur" =~ /elf(en)?/
=> 3
irb> $&
=> "elf"
```

Durch einen senkrechten Strich werden (wie bei der BNF) Alternativen gekennzeichnet.

```
irb> "Spielfigur" =~ /i(e|g)/
=> 2
irb> $&
=> "ie"
irb> $1
=> "e"
```

Dieses Beispiel zeigt, dass immer der linkest mögliche Teilstring gematcht wird und die Verwendung einer weiteren Variablen `$1`, die speichert, welcher Teilstring gegen den Regulären Ausdruck im ersten Klammerpaar gematcht wurde.

Analog zu `$1` werden weitere Variablen belegt, wenn es mehr als ein Klammerpaar gibt. Diese werden dabei anhand ihrer ersten Klammer von links nach rechts durchnummeriert.

```
irb> "Spielfigur" =~ /(.....)((..)...)/
=> 0
irb> $1
=> "Spiel"
irb> $2
=> "figur"
irb> $3
=> "fi"
```

Schließlich gibt es noch Möglichkeiten Wiederholungen zu spezifizieren. Der Stern kennzeichnet eine optionale Wiederholung und das Plus-Zeichen eine mindestens einmal wiederholte Zeichenfolge.

```
irb> "Spielfigur" =~ /pi.*g/
=> 1
irb> $&
=> "pielfig"
irb> "Spielfigur" =~ /pi.+g/
=> 1
irb> $&
=> "pielfig"
irb> "Spielfigur" =~ /pi.*e/
=> 1
irb> $&
```

```
=> "pie"
irb> "Spielfigur" =~ /pi.+e/
=> nil
```

Statt Teilzeichenketten zu suchen, können wir diese auch ersetzen. Die Methode `sub` auf Zeichenketten ersetzt das erste Vorkommen, die Methode `gsub` (`g` steht hier für *global*) ersetzt alle Vorkommen, einer passenden Zeichenkette.

```
irb> "Spielfigur".sub(/i.../,"lunk")
=> "Splunkigur"
irb> "Spielfigur".gsub(/i.../,"lunk")
=> "Splunklunk"
```

Von beiden Methoden stehen destruktive Varianten mit Ausrufezeichen zur Verfügung, die das Objekt, auf dem sie aufgerufen werden, verändern und zurückgeben, statt ein neues Objekt zu erzeugen.

17

Backtracking

Backtracking ist eine Programmiertechnik zum Lösen von Suchproblemen. Solche Probleme treten in der Praxis auf vielfaltige Weise auf. Zum Beispiel bei der konfliktfreien Zuteilung von Lehrkräften auf Schulklassen oder auch in kombinatorischen Puzzles auf der Rätselseite von Zeitschriften.

Haufig kann man bei Suchproblemen Teillösungen betrachten und schrittweise zu einer Problemlösung erweitern. Unterschiedliche Teillösungen bieten oft unterschiedliche Möglichkeiten, sie zu erweitern, so dass der sogenannte **Suchraum** aller (Teil-)Lösungen als Baumstruktur aufgefasst werden kann. Die Blätter dieses Baumes sind Teillösungen, die nicht mehr erweitert werden können. Diese können erfolgreiche Problemlösungen darstellen oder Fehlschläge. Die inneren Knoten des Baumes entsprechen Teillösungen und deren Nachfolgeknoten entsprechen ihren Erweiterungen.

Beim Lösen eines Sudoku-Puzzles zum Beispiel, entsprechen vollständig ausgefüllte Puzzles den Blättern im Suchbaum und konfliktfrei ausgefüllte Puzzles den Lösungen. Die inneren Knoten sind teilweise ausgefüllte Puzzles, die dadurch erweitert werden können, dass ein bisher freies Feld mit einer Ziffer belegt wird.

Die Verzweigungen im Suchbaum entstehen durch Alternativen bei der Erweiterung von Teillösungen und Backtracking ist eine Technik, diese Alternativen systematisch auszuprobieren. Dazu merkt man sich bei der Auswahl einer Alternative, welche weiteren Alternativen es gibt. Bei einem Fehlschlag nimmt man dann die zuletzt vorgenommene Auswahl zurück und probiert stattdessen die nächste Alternative. Diese Rückkehr zur zuletzt betrachteten Alternative gibt der Programmiertechnik Backtracking ihren Namen.

Ein Algorithmus, der mit Hilfe von Backtracking nach der ersten Lösung eines Problems sucht, kann durch Kombination einer Schleife mit Rekursion formuliert werden. Er gibt zurück,

ob eine Teillösung lösbar ist und muss mit einer initialen Teillösung aufgerufen werden.

```
Teillösung lösbar?
    Falls Teillösung vollständig ist,
        gib zurück, ob Teillösung gültig ist.

    Durchlaufe jede Erweiterung der Teillösung.
        Falls Erweiterung lösbar,
            gib wahr zurück.

    Gib falsch zurück.
```

Die Schleife, die die Erweiterungen einer Teillösung durchläuft, enthält in ihrem Rumpf einen rekursiven Aufruf des Algorithmus, um zu testen, ob die Erweiterungen lösbar sind. Falls keine der Erweiterungen lösbar ist, ist die Teillösung auch nicht lösbar, in diesem Fall wird also falsch zurück gegeben. Dieser Algorithmus basiert auf Unter-Algorithmen

- zum Testen, ob eine (Teil-)lösung vollständig ist,
- zum Testen, ob eine (Teil-)lösung gültig ist und
- zur Berechnung aller Erweiterungen einer Teillösung.

Unterschiedliche Backtracking-Algorithmen unterscheiden sich im Wesentlichen in diesen drei Aspekten, während das Grundgerüst gleich bleibt.

Damenproblem

Das Damenproblem ist ein einfach zu beschreibendes Problem, das sich elegant durch Backtracking lösen lässt und dabei erlaubt, die beschriebenen Aspekte eines Suchproblems zu verdeutlichen. Es besteht darin, acht Damen so auf einem Schachbrett zu platzieren, dass sie sich weder horizontal noch vertikal noch diagonal schlagen können. Im Folgenden ist eine gültige Platzierung von vier Damen auf einen entsprechend verkleinerten Schachbrett dargestellt.

Um die Platzierung von Damen auf einem Schachbrett in Ruby darzustellen, können wir ausnutzen, dass diese, damit sie sich nicht horizontal schlagen können, in unterschiedlichen Reihen platziert werden müssen. Wir stellen sie deshalb als Array aus Zahlen dar und speichern dabei im ersten Eintrag des Arrays, in welcher Spalte die erste Dame steht, im zweiten Eintrag die Spalte der zweiten Dame und so weiter. Zum Beispiel entspricht das Array `[2,0,3,1]` der folgenden Platzierung von vier Damen auf einem 4x4 Schachbrett.

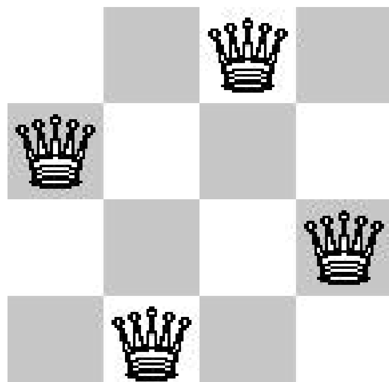


Abbildung 17.1: Vier Damen auf einem 4x4 Schachbrett, die sich nicht schlagen.

Die Prozedur `print_queens` gibt eine so dargestellte Platzierung von Damen im Terminal aus.

```
def print_queens queens
    queens.each do |q|
        puts("  "*q + "Q")
    end
end
```

Die Ausgabe von `print_queens [2,0,3,1]` ähnelt der obigen grafischen Darstellung.

```
    Q
Q
      Q
  Q
```

Um das Damenproblem mit Hilfe von Backtracking zu lösen, implementieren wir eine Funktion `complete?`, die testet, ob eine so dargestellte Platzierung vollständig ist. Da wir acht Damen auf einem richtigen Schachbrett platzieren wollen, testen wir dazu, ob das Array, die Größe acht hat. Unser Algorithmus soll also dem Array schrittweise Einträge hinzufügen, bis dieses acht Einträge enthält.

```
def complete? queens
    return queens.size == 8
end
```

Um zu testen, ob eine Platzierung gültig ist, müssen wir testen, ob alle Damen vor Angriffen anderer sicher sind. Da wir durch die Darstellung bereits sicher gestellt haben, dass Damen sich nicht horizontal schlagen können, brauchen wir dazu nur noch zu testen, ob sie sich vertikal oder diagonal bedrohen. Dazu durchlaufen wir jede Dame mit Hilfe einer Zählschleife und testen dann in einer weiteren Zählschleife, ob sie vor später platzierten Damen sicher ist.

```
def safe? queens
    safe = true

    for i in 0..queens.size-1 do
        q_i = queens[i]

        for j in (i+1)..queens.size-1 do
            q_j = queens[j]

            vertical = q_i == q_j
            diagonal = (q_i-q_j).abs == j - i

            if vertical || diagonal then
                safe = false
            end
        end
    end

    return safe
end
```

Ob sich Damen vertikal bedrohen, erkennen wir daran, ob sich die Spalten zweier verschiedener Damen gleichen. Um zu testen, ob sich Damen diagonal bedrohen, vergleichen wir deren Spaltenabstand mit dem Zeilenabstand. Sind diese gleich, stehen die Damen auf der selben Diagonale und können sich schlagen. Zur Berechnung des Spaltenabstandes verwenden wir den Absolutbetrag. Da wir für jede in der äußeren Schleife durchlaufene Dame nur später platzierte Damen betrachten, ist dies für den Zeilenabstand nicht nötig.

Schließlich implementieren wir noch eine Funktion `place_next`, die eine Platzierung um eine weitere Dame erweitert. Diese fügt dem Array einen Eintrag zwischen null und sieben hinzu und gibt ein Array aller so erzeugten Arrays zurück.

```
def place_next queens
    nums = [0,1,2,3,4,5,6,7]
    return nums.collect { |n| queens + [n] }
end
```

Wir können nun den zuvor umgangssprachlich formulierten Algorithmus als Ruby-Funktion `solvable?` implementieren. Statt im Schleifenrumpf eine `return`-Anweisung zu verwenden, speichern wir in einer Variablen `solvable`, ob eine Lösung gefunden wurde. Diese können wir dann in der Bedingung einer bedingten Schleife abfragen, um die Betrachtung überflüssiger Alternativen zu vermeiden.

```
def solvable? queens
    if complete? queens then
        return safe? queens
    end

    qs = place_next queens

    index = 0
    solvable = false
    while !solvable && index < qs.size do
        solvable = solvable? qs[index]
        index = index + 1
    end

    return solvable
end
```

Falls die übergebene Teillösung vollständig ist, wird zurückgegeben, ob diese gültig ist. Falls nicht, werden alle Erweiterungen der übergebenen Teillösung berechnet und in der Variablen `qs` gespeichert. Die anschließende Schleife durchläuft die Erweiterungen, bis mit Hilfe eines rekursiven Aufrufs eine lösbare Erweiterung gefunden wurde.

Mit den gezeigten Definitionen läuft der Aufruf `solvable? []` für einige Sekunden und liefert schließlich das Ergebnis `true` zurück, zeigt also an, dass das Damenproblem für acht Damen lösbar ist. Dabei werden alle Platzierungen von acht Damen auf einem Schachbrett nacheinandner daraufhin getestet, ob sich Damen bedrohen, bis die erste sichere Platzierung gefunden wurde. Da (bis zur ersten Lösung) der komplette Suchraum aller Platzierungen von acht Damen auf einem Schachbrett durchsucht wird, spricht man von einem sogenannten *brute force* Algorithmus. Der Suchraum wird mit voller Kraft vorraus aber auch blind durchsucht und erst vollständige Platzierungen werden auf Gültigkeit überprüft.

Da wir bereits unvollständige Teillösungen auf Gültigkeit überprüfen können, können wir die Laufzeit des Algorithmus deutlich verbessern. Wenn sich zum Beispiel schon die beiden zuerst platzierten Damen bedrohen, brauchen die restlichen sechs gar nicht mehr platziert zu werden. Dadurch werden große Teile des Suchbaums gar nicht erst durchlaufen. Wir implementieren diese Idee, indem wir nur *gültige* Erweiterungen einer Teillösung rekursiv testen. Dazu fügen wir nach der Zuweisung an die Variable `qs` die folgende Zeile ein.

```
    qs = qs.select { |q| safe? q }
```

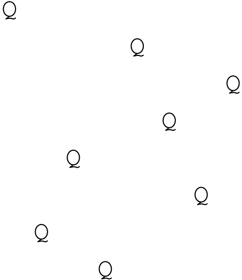
Nach dieser Änderung liefert der Aufruf `solvable? []` das Ergebnis `true` ohne merkliche Verzögerung. Falls wie hier be-

reits *Teil*lösungen auf ihre Gültigkeit überprüft werden können, kann auf diese Weise die Laufzeit des Backtracking-Verfahrens oft erheblich verbessert werden.

Dass das Damenproblem lösbar ist, haben wir möglicherweise bereits vorher vermutet. Um die gefundene Platzierung auszugeben, fügen wir der bedingten Anweisung zu Beginn der Definition von `solvable?` eine entsprechende Zeile hinzu.

```
if complete? queens then
    print_queens queens
    return true
end
```

Da wir `solvable?` nur noch mit gültigen Teillösungen aufrufen, ist der Test `safe? queens` innerhalb der bedingten Anweisung nun überflüssig und wir ersetzen ihn durch `true`. Der Aufruf `solvable? []` erzeugt nun die folgende Darstellung einer sicheren Platzierung von acht Damen auf einem Schachbrett.



Quellen uns Lesetipps

- Backtracking-Einführung der University of Pennsylvania
- Damenproblem bei Wikipedia

18

Künstliche Intelligenz für Spiele

Im Allgemeinen werden zwei Arten, Künstliche Intelligenz zu verwirklichen unterschieden: klassische Programmierung und Maschinelles Lernen. Spiele sind ein Anwendungsfeld der Künstlichen Intelligenz, in dem klassische Programmierung lange erfolgreich eingesetzt wurde. Ähnlich wie beim Backtracking, macht man sich dazu zu nutze, dass Computer es erlauben, viele Möglichkeiten, wie sich ein Spiel entwickeln kann, zu durchsuchen und zu vergleichen. In diesem Kapitel lernen wir Algorithmen kennen, die auf diese Weise arbeiten.

Zwei-Personen-Spiele und automatisches Spiel

Im folgenden betrachten wir Spiele, in denen zwei Spieler abwechselnd ziehen, (anders als bei vielen Kartenspielen) keine Information geheim bleibt und (anders als bei Würfelspielen) der Zufall keine Rolle spielt. Beispiele für solche Spiele sind Schach, Dame, Reversi, Tic Tac Toe oder Vier Gewinnt.

Wir werden Ruby-Klassen definieren, die es erlauben Zwei-Personen-Spiele darzustellen und Klassen, die es erlauben, automatische Spieler für solche Spiele zu definieren. Da die Algorithmen unabhängig von den konkreten Spielen definiert werden können, trennen wir die Definition von Spielern von der Definition von Spielen.

Spieler für Zwei-Personen-Spiele sind Objekte der Klasse `Player`.

```
class Player
  def initialize name
    @name = name
  end

  def game= game
    @game = game
  end
```

```
  def to_s
    return @name
  end
end
```

Zur Darstellung auf dem Bildschirm geben wir Spielern einen Namen, der im Konstruktor übergeben wird. Unterklassen der Klasse `Player` sollen eine Methode `select_move` implementieren, die einen ausgewählten Zug im Spiel zurückliefert, dass im Attribut `@game` gespeichert ist. Unterklassen der Klasse `Game` sollen dazu eine Methode `valid_moves` definieren, die ein Array gültiger Züge liefert, aus dem Spieler wählen können.

Zwei-Personen-Spiele sind Objekte der Klasse `Game`:

```
class Game
  def initialize(player1, player2)
    @current_player = player1
    @current_player.game = self

    @waiting_player = player2
    @waiting_player.game = self
  end

  # weitere Definitionen folgen

end
```

Objekten der Klasse `Game` werden im Konstruktor zwei Spieler übergeben. Der zuerst übergebene Spieler beginnt das Spiel, und nach seinem Zug wechselt das Zugrecht. Dazu vertauscht die Methode `next_turn!` die Rollen beider Spieler.

```
  def next_turn!
    player = @current_player
    @current_player = @waiting_player
    @waiting_player = player
  end
```

Die Methode `play!` wird aufgerufen um ein Spiel zu starten und seine Durchführung auf dem Bildschirm auszugeben.

```
  def play!
    while !ended? do
      puts self
      make_move! @current_player.select_move
      next_turn!
    end
    puts self
  end
```

In jedem Schritt wird ein von dem Spieler, der an der Reihe ist, ausgewählter Zug ausgeführt, bis das Spiel beendet ist. Dazu müssen die Methoden `ended?` und `make_move!` von Unterklassen der Klasse `Game` implementiert werden.

Vor und nach jedem Zug wird das Spiel auf dem Bildschirm ausgegeben. Die Methode `to_s` gibt eine Zeichenkette zurück, die den Zustand des Spiels beschreibt.

```
def to_s
  if ended? then
    w = winner
    if w == nil then
      return "It's a draw"
    else
      return w.to_s + " wins"
    end
  end
  return @current_player.to_s + "'s turn"
end
```

Ist das Spiel beendet, so wird mit Hilfe der Methode `winner` ermittelt, wer gewonnen hat, und das Ergebnis zurück geliefert. Auch diese Methode muss also in Unterklassen implementiert werden. Läuft das Spiel noch, liefert `to_s` zurück, wer an der Reihe ist.

Die gezeigten Definitionen der Klassen `Player` und `Game` speichern wir in `two_player_games.rb`, um sie zur Definition konkreter Spiele und Spieler in anderen Dateien verwenden zu können.

Als Beispiel für ein einfaches Zwei-Personen-Spiel implementieren eine einfache Version des Nim-Spiel's als Unterklasse von `Game` in der Date `simple_nim.rb`.

```
require "./two_player_games.rb"

class SimpleNim < Game
  def initialize(player1, player2, count)
    super(player1, player2)
    @count = count
  end

  # weitere Definitionen folgen

end
```

Das Nim-Spiel wird mit einem Haufen Streichhölzern gespielt, dessen Größe im Konstruktor übergeben wird. Die Methode

`to_s` gibt die Anzahl der Streichhölzer neben dem Spielzustand zurück, den die überschriebene Methode der Oberklasse liefert.

```
  def to_s
    return (@count.to_s + "\tmatches, " + super)
  end
```

Die Spieler nehmen abwechselnd Streichhölzer vom Haufen, bis keine mehr da sind.

```
  def ended?
    return @count == 0
  end
```

Die Methode `make_move!` entfernt so viele Streichhölzer vom Haufen, wie im übergebenen Zug angegeben sind.

```
  def make_move! move
    @count = @count - move
  end
```

Wer das letzte Streichholz nimmt, verliert das Spiel. Es gewinnt also der Spieler, der bei Spielende an der Reihe ist.

```
  def winner
    return @current_player
  end
```

Ein gültiger Zug entfernt ein bis drei Streichhölzer vom Haufen, sofern noch so viele dort liegen. Die Höchstzahl zu entfernender Streichhölzer wird mit der Methode `min` der Klasse `Array` berechnet.

```
  def valid_moves
    moves = []
    for move in 1..[@count,3].min do
      moves.push move
    end
    return moves
  end
```

Um unsere Implementierung zu testen, definieren wir in der Datei `random_player.rb` eine Klasse für Spieler, die in jedem Zug einen zufälligen der gültigen Züge auswählen.

```
require "./two_player_games.rb"

class RandomPlayer < Player
```

```
  def select_move
    return @game.valid_moves.shuffle.first
  end
end
```

Wir können nun mit `irb` ein Spiel zwischen Zufallsspielern starten und den Verlauf beobachten.

```
irb> alice = RandomPlayer.new("Alice")
irb> bob = RandomPlayer.new("Bob")
irb> SimpleNim.new(alice,bob,21).play!
21  matches, Alice's turn
20  matches, Bob's turn
18  matches, Alice's turn
17  matches, Bob's turn
15  matches, Alice's turn
14  matches, Bob's turn
11  matches, Alice's turn
8   matches, Bob's turn
5   matches, Alice's turn
4   matches, Bob's turn
2   matches, Alice's turn
0   matches, Bob wins
```

Bewertung von Spielzügen

Um gute Spieler zu programmieren, müssen wir statt zufälligen sinnvolle Züge aus den gültigen auswählen. Diese Auswahl ist der Kern der in diesem Kapitel vorgestellten Algorithmen, die wir im folgenden schrittweise entwickeln.

Vollständige Suche im Spielbaum

In einfachen Spielen können wir alle Zugmöglichkeiten systematisch überprüfen, indem wir (ähnlich wie beim Backtracking) alle Folgezüge durchsuchen, bis das Spiel beendet ist. Dadurch entsteht eine Baumstruktur, an deren Blättern beendete Spiele stehen. Jeder innere Knoten verzweigt entsprechend der in diesem Zustand gültigen Züge.

Statt bei jeder Verzweigung Kopien der Spielzustände anzulegen, wollen wir das Spiel-Objekt mutieren. Dazu müssen Spiele neben der Methode `make_move!` eine Methode `undo_move!` definieren, die einen übergebenen Zug rückgängig macht. Dann können Züge probeweise mit `make_move!` ausgeführt und vor dem Ausprobieren weiterer Züge mit `undo_move!` wieder rückgängig gemacht werden.

Für das Nim-Spiel definieren wir die Methode `undo_move!` wie folgt.

```
def undo_move! move
  @count = @count + move
end
```

Die im übergebenen Zug herunter genommen Streichhölzer werden hier also wieder auf den Haufen drauf gelegt.

Verglichen mit Backtracking kommt bei Spielbäumen erschwerend hinzu, dass zwei Spieler mit unterschiedlichen Zielen gegeneinander antreten. Was für den einen Spieler ein günstiger Zug ist, ist für den anderen Spieler ein ungünstiger. Beide Spieler versuchen, Züge so auszuwählen, dass sie ein möglichst gutes Ergebnis erzwingen können. Ist kein Sieg erzwingbar, kann möglicherweise zumindest ein Unentschieden gesichert werden, um eine Niederlage zu vermeiden.

Für Spiele im Endzustand können wir diese drei Ergebnisse als Gewinnwahrscheinlichkeit (1 für einen Sieg, $0,5$ für ein Unentschieden und 0 für eine Niederlage) ausdrücken. Zur Berechnung dieser Gewinnwahrscheinlichkeit (für den Spieler der an der Reihe ist) fügen wir der Klasse `Game` die folgende Methode hinzu.

```
def final_chances
  if winner == @current_player then
    return 1
  end
  if winner == @waiting_player then
    return 0
  end
  return 0.5
end
```

Wir definieren nun eine Klasse `SearchingPlayer` für Spieler, die den Spielbaum systematisch bis zum Ende durchsuchen. Die vier ersten Methoden können später von Unterklassen überschrieben werden, um die Suche zu beeinflussen.

```
class SearchingPlayer < Player
  def stop_search?
    return @game.ended?
  end

  def guess_chances
    return @game.final_chances
  end
```

```
  def make_move!(move, branch_count)
    @game.make_move! move
    @game.next_turn!
  end

  def undo_move!(move, branch_count)
    @game.undo_move! move
    @game.next_turn!
  end

  # weitere Definitionen folgen

end
```

In der Klasse `SearchingPlayer` sind die gezeigten Methoden durch bereits besprochene Methoden auf Spielen implementiert. Die teilweise abweichenden Namen und Parameter klären wir später.

Die wichtigste Methode eines Spielers ist die Methode `select_move`. Hier berechnet sie das Maximum aller Gewinnwahrscheinlichkeiten und gibt einen entsprechenden Zug zurück.

```
  def select_move
    moves = @game.valid_moves

    if moves.size == 1 then
      return moves.first
    end

    best_move = nil
    best_chances = -1
    moves.each do |move|
      make_move!(move, moves.size)
      chances = 1 - winning_chances
      undo_move!(move, moves.size)

      if chances > best_chances then
        best_move = move
        best_chances = chances
      end
    end
    return best_move
  end
```

Falls es nur einen einzigen gültigen Zug gibt, wird dieser zurück gegeben. Ansonsten werden alle gültigen Züge der Reihe nach durchsucht. Für jeden Zug wird eine Gewinnwahrscheinlichkeit berechnet, und am Ende wird der Zug mit der höchsten

Gewinnwahrscheinlichkeit zurück gegeben. Die Gewinnwahrscheinlichkeit ergibt sich als umgekehrte Gewinnwahrscheinlichkeit des Gegners, der nach unserem Zug an der Reihe ist. Kann der Gegner einen Sieg erzwingen, ist uns eine Niederlage sicher (und umgekehrt).

Der in der Methode `winning_chances` implementierte Algorithmus berechnet den bestmöglichen Ausgang unter der Annahme, dass beide Spieler versuchen, ihre eigene Gewinnwahrscheinlichkeit zu maximieren.

```
def winning_chances
  if stop_search? then
    return guess_chances
  end

  moves = @game.valid_moves
  return moves.collect do |move|
    make_move!(move, moves.size)
    chances = 1 - winning_chances
    undo_move!(move, moves.size)
    chances
  end.max
end
```

Falls `stop_search?` angibt, dass die Suche abgebrochen werden soll, wird das Ergebnis von `guess_chances` zurück geliefert. Mit den oben gezeigten Implementierungen wird also die Suche abgebrochen, wenn das Spiel beendet ist und dann eine dem Spielausgang entsprechende Gewinnwahrscheinlichkeit zurück gegeben.

Soll weiter gesucht werden (ist also das Spiel noch nicht beendet) werden ähnlich wie schon in `select_move` alle gültigen Züge durchsucht. Hier werden nur deren Gewinnwahrscheinlichkeiten berechnet, von denen dann das Maximum zurück gegeben wird. Wie oben ergeben sich die Gewinnwahrscheinlichkeiten der Folgezüge als umgekehrte Gewinnwahrscheinlichkeiten des Gegners.

Wenn wir nun eine Instanz der Klasse `SearchingPlayer` im Nim-Spiel gegen einen zufälligen Spieler antreten lassen, sollte in der Regel der zufällige Spieler verlieren. Hier ist eine entsprechende Beispielausgabe.

```
irb> alice = SearchingPlayer.new "Alice"
irb> bob = RandomPlayer.new "Bob"
irb> SimpleNim.new(alice,bob,21).play!
21  matches, Alice's turn
20  matches, Bob's turn
18  matches, Alice's turn
```

```
17  matches, Bob’s turn
14  matches, Alice’s turn
13  matches, Bob’s turn
10  matches, Alice’s turn
9   matches, Bob’s turn
7   matches, Alice’s turn
5   matches, Bob’s turn
3   matches, Alice’s turn
1   matches, Bob’s turn
0   matches, Alice wins
```

Für größere Streichholzhaufen können wir beobachten, dass die Suche nach dem besten Zug sehr lange dauert.

Tiefenbeschränkte Suche im Spielbaum

Für komplexe Spiele ist es nicht praktikabel, den Spielbaum vollständig zu durchsuchen. Es ist daher üblich, Zugfolgen nicht bis zum Ende sondern nur bis zu einer bestimmten Tiefe im Baum zu verfolgen. Die Klasse `LimitingPlayer` definiert dazu ein Attribut `@limit` für die maximale Anzahl von Verzweigungen, die bei einer durchsuchten Zugfolge durchlaufen werden dürfen.

```ruby
require "./searching_player.rb"

class LimitingPlayer < SearchingPlayer
  def initialize(name, limit)
    super name
    @limit = limit
  end

  # weitere Definitionen folgen

end
```

Um die Suchtiefe wie beschrieben zu beschränken, überschreiben wir die Methode `stop_search?` wie folgt.

```ruby
  def stop_search?
    return @limit == 0 || super
  end
```

Da die Suche nun möglicherweise bei einem Spiel abbricht, das noch nicht beendet ist, müssen wir die Methode `guess_chances` so anpassen, dass sie auch mit nicht beendeten Spielen zurecht kommt.

```
def guess_chances
  if @game.ended? then
    return super
  else
    return (1.0 + rand(9)) / 10
  end
end
```

Falls das Spiel beendet ist, rufen wir die dafür ausgelegte Implementierung der Oberklasse auf und geben ihr Ergebnis zurück. Falls nicht, geben wir eine zufällige Gewinnwahrscheinlichkeit zwischen $0,1$ und $0,9$ zurück. Für Spiele, für die wir eine spezialisierte Bewertungsfunktion angeben können, können wir eine Unterklasse von `LimitingPlayer` definieren, in der wir die Methode `guess_chances` überschreiben.

Die Methoden `make_move!` und `undo_move!` überschreiben wir so, dass das Attribut `@limit` manipuliert wird, wenn es Alternativen zu dem übergebenen Zug gibt.

```
def make_move!(move, branch_count)
  super(move, branch_count)
  if branch_count > 1 then
    @limit = @limit - 1
  end
end

def undo_move!(move, branch_count)
  super(move, branch_count)
  if branch_count > 1 then
    @limit = @limit + 1
  end
end
```

Mit diesen Definitionen, implementieren die geerbten Methoden `select_move` und `winning_chances` die beschriebene tiefenbeschränkte Suche. Wir können damit eine weitere Simulation des Nim-Spiels starten.

```
irb> alice = LimitingPlayer.new("Alice",10)
irb> bob = RandomPlayer.new "Bob"
irb> SimpleNim.new(alice,bob,42).play!
42  matches, Alice's turn
41  matches, Bob's turn
40  matches, Alice's turn
39  matches, Bob's turn
37  matches, Alice's turn
36  matches, Bob's turn
35  matches, Alice's turn
```

```
34  matches, Bob's turn
33  matches, Alice's turn
32  matches, Bob's turn
30  matches, Alice's turn
29  matches, Bob's turn
28  matches, Alice's turn
27  matches, Bob's turn
25  matches, Alice's turn
24  matches, Bob's turn
23  matches, Alice's turn
22  matches, Bob's turn
21  matches, Alice's turn
18  matches, Bob's turn
16  matches, Alice's turn
13  matches, Bob's turn
11  matches, Alice's turn
9   matches, Bob's turn
6   matches, Alice's turn
5   matches, Bob's turn
4   matches, Alice's turn
1   matches, Bob's turn
0   matches, Alice wins
```

Trotz der beschränkten Suchtiefe gelingt es Alice am Ende gegen den zufälligen Spieler Bob zu gewinnen.

Beschneidung des Spielbaums

Bei geschickter Protokollierung von Zwischenergebnissen, gibt es noch mehr Potential, die Suche im Spielbaum vorzeitig abzubrechen. Bei der Suche im Spielbaum mit der Methode `select_move` speichern wir als Zwischenergebnis den Wert `best_chances` für die beste bisher gefundene Gewinnwahrscheinlichkeit. Rekursive Aufrufe speichern entsprechende Werte für uns und den Gegner. Aus diesen Werten ergeben sich Grenzen für solche Gewinnwahrscheinlichkeiten, die für die Suche interessant sind. Gewinnwahrscheinlichkeiten, die unterhalb dem bisher gefundenen besten Wert liegen, sind uninteressant, weil sie weniger Erfolg versprechen als ein bereits gefundener Zug. Statt dem niedrigeren Wert können wir gefahrlos den bisher besten gefundenen Wert zurückgeben, ohne das Ergebnis der Suche zu beeinflussen, denn der beste Wert wird nur bei einem noch besseren Wert angepasst.

Interessanterweise lässt sich auch eine Obergrenze für interessante Gewinnwahrscheinlichkeiten angeben. Gewinnwahrscheinlichkeiten, die oberhalb der Obergrenze liegen, sind uninteressant, wenn der Gegner bereits eine Möglichkeit gefunden

hat, uns eine niedrigere Gewinnwahrscheinlichkeit aufzuzwingen. Die Obergrenze ergibt sich also aus der bisherigen besten Gewinnwahrscheinlichkeit des Gegners. Sobald uns eine Zugmöglichkeit zur Verfügung steht, die die Obergrenze überschreitet, können wir die Suche abbrechen, weil wir davon ausgehen können, dass der Gegner den vorherigen Zug, der uns diese Möglichkeiten bescherte, nicht auswählen wird. Statt des größeren Wertes können wir gefahrlos die übergebene Obergrenze zurück liefern, ohne das Ergebnis der Suche zu verändern, weil der Gegner einen bisher gefundenen Wert nur anpasst, wenn er uns eine noch niedrigere Gewinnwahrscheinlichkeit aufzwingen kann.

Die Klasse `PruningPlayer` überschreibt die Methode `winning_chances` unter Verwendung zusätzlicher Parameter für die besprochenen Grenzen.

```
require "./limiting_player.rb"

class PruningPlayer < LimitingPlayer
  def winning_chances(min, max)
    if stop_search? then
      return guess_chances
    end

    index = 0
    moves = @game.valid_moves
    best_chances = min
    while best_chances < max && index < moves.size do
      make_move!(moves[index], moves.size)
      chances = 1 - winning_chances(1-max, 1-best_chances)
      undo_move!(moves[index], moves.size)
      index = index + 1

      if chances > best_chances then
        best_chances = chances
      end
    end

    return best_chances
  end

  # weitere Definition folgt

end
```

Die Definition von `winning_chances` ähnelt der vorherigen Definition. Statt einer Zählschleife wird jedoch eine bedingte Schleife verwendet, die abgebrochen wird, sobald ein Zug

gefunden wurde, dessen Gewinnwahrscheinlichkeit die Obergrenze übersteigt. Im rekursiven Aufruf wird als Obergrenze für den Gegner die umgekehrte bisher beste eigene Gewinnwahrscheinlichkeit übergeben. Analog dazu wird als Untergrenze die umgekehrte Obergrenze verwendet, die dem bisher besten gefundenen Zug des Gegners entspricht.

Die neue Implementierung von `select_move` unterscheidet sich von der ursprünglichen nur durch den Aufruf von `winning_chances`.

```
def select_move
  moves = @game.valid_moves

  if moves.size == 1 then
    return moves.first
  end

  best_move = nil
  best_chances = -1
  moves.each do |move|
    make_move!(move, moves.size)
    chances = 1 - winning_chances(-1, 1-best_chances)
    undo_move!(move, moves.size)

    if chances > best_chances then
      best_move = move
      best_chances = chances
    end
  end
  return best_move
end
```

Als Bereichsgrenzen übergeben wir solche außerhalb der berechneten Gewinnwahrscheinlichkeiten. Die Untergrenze ist so klein, dass sie durch den ersten gefundenen Zug angehoben wird. Die Obergrenze ist so groß, dass sie durch den ersten gefundenen Zug des Gegners abgesenkt wird.

Nach diesen Anpassungen liefert die neue Implementierung von `select_move` den gleichen Zug wie die ursprüngliche. Dabei werden weniger Zugfolgen betrachtet als mit der ursprünglichen Implementierung, weil Teile des Spielbaums, die das Ergebnis nicht beeinflussen, abgeschnitten werden. Instanzen der Klasse `PruningPlayer` verhalten sich also wie Instanzen von `LimitingPlayer`, berechnen ihren Zug aber schneller. Die Suche ist weiterhin tiefenbeschränkt, und spezialisierte Implementierungen von `guess_chances` können in Unterklassen definiert werden.

Quellen

- Minimax-Suche in der englischen Wikipedia
- Alpha-Beta-Pruning in der englischen Wikipedia
- Erklärung des Alpha-Beta-Algorithmus der RWTH-Aachen

19
Funktionale Programmierung

Funktionale Programmierung heißt *funktional*, weil Programme nur[1] aus Funktionen bestehen. Das Hauptprogramm selbst ist eine Funktion, die die Eingabe des Programms als Argument nimmt und dessen Ausgabe als Ergebnis liefert. In der Regel verwendet das Hauptprogramm Hilfsfunktionen, die in Ihrer Definition ihrerseits andere Funktionen verwenden, bis schließlich primitive Funktionen verwendet werden.

[1] Diese Behauptung ignoriert scheinbar Ein- und Ausgabe, deren Integration in rein Funktionale Programme wir hier nicht vertiefen.

Die Vorteile Funktionaler Programmierung könnten durch Weglassen von Charakteristika traditioneller imperativer Programmierung beschrieben werden. So gibt es in Funktionalen Programmen keine Zuweisungen; der Wert von Variablen kann also nicht verändert werden. Nicht nur die Zuweisung kommt in Funktionalen Programmen nicht vor, sondern jene enthalten überhaupt keine sogenannten Seiteneffekte. Es gibt also keine Anweisungen sondern nur Ausdrücke und das Ergebnis einer Funktion ist allein durch die Werte ihrer Argumente beschrieben. Dadurch wird die Auswertungsreihenfolge Funktionaler Programme irrelevant und Teilausdrücke können durch ihren Wert ersetzt werden, ohne das Verhalten eines Programms zu ändern.

Solche Einschränkungen als Vorteil zu verkaufen greift jedoch zu kurz, denn welchen Vorteil sollte der Verzicht auf bestimmte Sprachkonstrukte für die Entwicklung von Programmen haben? Tatsächlich gab es eine ähnliche Situation schon einmal, als machinennahe Programmierung mit beliebigen Sprüngen zum Beispiel durch `GOTO`-Anweisungen durch sogenannte *Strukturierte Programmierung* abgelöst wurde. Auch damals wurde der Verzicht auf bestimmte Konstrukte als Vorteil verkauft. Tatsächlich war es kein Verzicht sondern ein neuer Fokus, der den Kern Strukturierter Programmierung ausmachte. Die Bevorzugung komplexer Kontrollstrukturen wie Schleifen und bedingter Anweisungen gegenüber einfachen bedingten Sprüngen und insbesondere die Verwendung prozeduraler Abstraktion führten zu erhöhter Modularisierung von Programmen, die deren Entwicklung vereinfachte.

Der Fokus Funktionaler Programmierung auf Funktionen führt zu neuen Arten einfache Funktionen zu komplexeren zu komponieren und schafft auf diese Weise ebenfalls neue Möglichkeiten Programme zu modularisieren. Besonders hilfreich sind hierzu sogenannte *Funktionen höherer Ordnung*, die andere Funktionen als Argumente oder Ergebnis haben. Einige Aspekte Funktionaler Programmierung wurden mittlerweile in imperative Sprachen integriert, so dass es, allerdings mit leicht erhöhtem Aufwand, zum Beispiel auch in Ruby möglich ist, funktional zu Programmieren.

Ein weiterer wichtiger Aspekt einiger Funktionaler Programmiersprachen ist das Typsystem. Auch in Bezug auf Typisierung gehen die im Rahmen Funktionaler Programmierung entwickelten Konzepte über die traditioneller imperativer Sprachen hinaus und finden Einzug in neuere imperative Sprachen.

Historische Bemerkungen

Bevor wir beginnen funktional zu programmieren, ordnen wir das Feld der Funktionalen Programmierung geschichtlich ein. Zentrale Grundideen der Funktionalen Programmierung sind in der Vorlesung, die John Backus anlässlich seiner Auszeichnung mit dem Turing Award[2] hielt, enthalten. Backus wurde ausgezeichnet für seine

[2] Der Turing Award ist der "Nobelpreis der Informatik".

> profound, influential, and lasting contributions to the design of practical high-level programming systems, notably through his work on Fortran, and for seminal publication of formal procedures for the specifications of programming languages

Fortran war eine der ersten höheren Programmiersprachen. Mit den genannten "formal procedures" ist die Backus-Naur-Form (BNF) gemeint.

Statt darüber vorzutragen, wofür er ausgezeichnet wurde, entschied sich Backus in seinem Vortrag den Blick nach vorne zu richten. Er nannte seine Vorlesung:

> Can Programming Be Liberated from the von Neumann Style? A Functional Style and Its Algebra of Programs

Der erste Absatz enthält die zentralen Themen:

> Conventional programming languages are growing ever more enormous, but not stronger. Inherent defects at the most basic level cause them to be both fat and weak: their primitive word-at-a-time style of programming inherited from their common ancestor–the von Neumann computer, their close coupling of semantics to state transitions, their division of programming

> into a world of expressions and a world of statements, their inability to effectively use powerful combining forms for building new programs from existing ones, and their lack of useful mathematical properties for reasoning about programs.

Backus argumentiert in seiner Vorlesung, dass konventionelle Programmiersprachen zu sehr von der Hardware (des von Neumann Rechners), auf der sie ausgeführt werden, beeinflusst sind. Durch Problembeschreibungen, die sich an der Maschine orientieren, werden (laut Backus) Programmierer daran gehindert, ein ausgeprägtes Verständnis für Programme zu entwickeln.

Backus prägte in seiner Vorlesung den Begriff des "von Neumann Falschenhalses", der in einem Rechner den Speicher mit dem Rechenwerk verbindet. Komplexe Änderungen des Speichers müssen "Wort für Wort" durch diesen Flaschenhals mit dem Rechenwerk ausgetauscht werden. Konventionelle Programmierung bestehe im Wesentlichen darin, diesen Austausch "Wort für Wort" zu organisieren und verschleiere dadurch den Blick auf das große Ganze:

> Not only is this tube a literal bottleneck for the data traffic of a problem, but, more importantly, it is an intellectual bottleneck that has kept us tied to word-at-a-time thinking instead of encouraging us to think in terms of the larger conceptual units of the task at hand.

Heutzutage brauchen Daten nicht mehr wortweise gehandhabt zu werden. Komplexe Datentypen wie Arrays und Hashes erlauben es, Daten hierarchisch zu strukturieren. Funktions-Definitionen erlauben es komplexe Ausdrücke hierarchisch in Komponenten zu zerlegen. Ein ähnliches Mittel zur Strukturierung von Anweisungen existiert in konventionellen Programmiersprachen jedoch nicht. Zwar können durch Prozeduren Anweisungsfolgen abstrahiert werden, aber Definitionen, die über Anweisungen parametrisiert sind, lassen sich (in konventionellen Sprachen) nicht erstellen. Ruby erlaubt dies durch die Übergabe von Blöcken an Funktionen; ein Sprachmittel, das deutlich von Funktionaler Programmierung inspiriert ist.

Sprachmittel, die es erlauben, große Programme aus kleinen zusammen zu setzen, erlauben dadurch das große Ganze im Blick zu behalten. Solches Verständnis wird gefördert, wenn die Operationen, die verwendet werden um Programme zu kombinieren, Regeln gehorchen. Bereits aus der Schulmathematik kennen wir Gesetze, wie das Distributivgesetz, die wir anwenden können (ohne sie jedesmal neu zu beweisen), um über Ausdrücke nachzudenken. Analog dazu ist es erstrebenswert, über konstruierte Programme nachdenken zu können, ohne jedesmal die genaue Arbeitsweise verwendeter Funktionen einzubeziehen.

Wie könnten solche Gesetze aussehen? Die folgende Gleichung, die besagt, dass Aufrufe der Array-Methoden `collect` und `reverse` vertauscht werden können, erscheint auf den ersten Blick sinnvoll:

```
a.collect { |x| ... }.reverse
  == a.reverse.collect { |x| ... }
```

In Kombination mit der Gleichung `a.reverse.reverse == a` könnte sie verwendet werden, um komplexe Ausdrücke, die verschiedene Aufrufe von `collect` und `reverse` enthalten zu vereinfachen.

Leider gilt die oben gezeigte Gleichung nicht, wie das folgende Beispiel zeigt:

```
irb> n = 0
 => 0
irb> [41,42,43].collect { |x|
   >   if n == 0 then n = x else n end
   > }.reverse
 => [41, 41, 41]
irb> n = 0
 => 0
irb> [41,42,43].reverse.collect { |x|
   >   if n == 0 then n = x else n end
   > }
 => [43, 43, 43]
```

Das Problem ist hier, dass das Ergebnis des übergebenen Blocks von einer außerhalb definierten Variable abhängt, die vom Block verändert wird. Das solche Seiteneffekte durch Anweisungen in Ruby erlaubt sind (der Wert des Blockes also nicht nur von seinen Argumenten abhängt), macht es uns schwerer, Gesetze zu finden, die es erlauben, Programme zu verstehen.

Rein Funktionale Programme erlauben keine Seiteneffekte. In der rein Funktionalen Sprache Elm gilt zum Beispiel die (der obigen entsprechende) Gleichung

```
List.reverse (List.map f l)
  == List.map f (List.reverse l)
```

In rein Funktionalen Sprachen gibt es keine Anweisungen, und Ausdrücke haben keine Seiteneffekte. Funktionen sind kein gesondertes Sprachkonstrukt sondern (wie z.B. Zahlen und Wahrheitswerte) mögliche Werte von Ausdrücken, die von anderen Funktionen als Ergebnis geliefert oder als Argument erwartet werden können.

Ausdrücke und primitive Typen in Elm

Für Elm gibt es online eine interaktive Programmierumgebung, die wir analog zu `irb` zur interaktiven Auswertung von Ausdrücken verwenden können.

Wie in Ruby gibt es auch in Elm primitive Werte zur Darstellung von Zahlen, Wahrheitswerten und Zeichenketten sowie vordefinierte Funktionen, um mit Ihnen zu rechnen. Die folgenden Eingaben demonstrieren den Umgang mit solchen Werten.

```
elm> 1+2 /= 3
False : Bool
elm> True && not True
False : Bool
elm> 1+2 <= 4*7 || 1+2 /= 3
True : Bool
elm> "Hello " ++ "World!"
"Hello World!" : String
elm> String.length "Hallo"
5 : Int
```

Elm gibt zu jedem Ausdruck neben dem Wert hinter einem Doppelpunkt den Typ des Wertes aus. Wahrheitswerte haben den Typ `Bool`, Ganzzahlen den Typ `Int` und Zeichenketten den Typ `String`. Zur Konkatenation von Zeichenketten verwenden wir `++` statt `+` wie in Ruby. Ungleichheit wird mit dem Operator `/=` notiert statt mit `!=`. Im letzten Aufruf qualifizieren wir die Funktion `length` mit dem Namen des Moduls, aus dem sie kommt. `String.length` ist die `length`-Funktion, die im Modul `String` (das Hilfsfunktionen zum Umgang mit Zeichenketten enthält) definiert ist. Im Modul `List` ist eine andere `length`-Funktion definiert:

```
elm> List.length [1,2,3]
3 : Int
elm> List.length [True,False]
2 : Int
elm> List.length [[],[1,2,3],[4]]
3 : Int
elm> List.length [True,"False"]
-- TYPE MISMATCH ----------------------------------

The 1st and 2nd entries in this list are different
types of values.

3|               [True,"False"]
                       ^^^^^^^
The 1st entry has this type:
```

```
    Bool

But the 2nd is:

    String
```

Listen in Elm werden ähnlich notiert wie Arrays in Ruby, dürfen allerdings keine Elemente unterschiedlichen Typs enthalten.

Die gezeigte Schreibweise für Listen ist eine Kurzschreibweise für die folgenden Anwendungen des sogenannten *Cons*-Operators `(::)`.

```
elm> "hello"::[]
["hello"] : List String
elm> "hello"::("world"::("!"::[]))
["hello","world","!"] : List String
elm> "hello"::"world"::"!"::[]
["hello","world","!"] : List String
elm> "hello"::["world","!"]
["hello","world","!"] : List String
elm> ["hello"]++["world","!"]
["hello","world","!"] : List String
```

Cons hängt also ein einzelnes Element vorne an eine andere Liste an und wird rechtsassoziativ geklammert. Auch Listen können, wie wir sehen, mit Hilfe des Operators ++ konkateniert werden.

Um Werte unterschiedlichen Typs zusammenzufassen, gibt es in Elm Tupel und Records.

```
elm> ("Sebastian",76.5)
("Sebastian",76.5) : ( String, Float )
elm> Tuple.second ("Sebastian",76.5)
76.5 : Float
elm> { name = "Sebastian", weight = 76.5 }
{ name = "Sebastian", weight = 76.5 } : { name : String, weight : Float }
elm> { name = "Sebastian", weight = 76.5 }.weight
76.5 : Float
```

Wie wir sehen, enthält der Typ von Records Informationen über die Namen und Typen seiner Komponenten, auf die wir wie auf Methoden in Ruby mit der Punkt-Schreibweise zugreifen können. Tupel speichern keine Namen für die Komponenten, auf die wir stattdessen mit Funktionen aus dem `Tuple`-Modul zugreifen können.

Eigene Definitionen in Elm

Einfache Funktionen können wir direkt in der interaktiven Programmierumgebung definieren:

```
elm> inc n = n + 1
<function> : number -> number
elm> inc 41
42 : number
elm> add x y = x + y
<function> : number -> number -> number
elm> add 47 11
58 : number
```

Die Funktion `inc` addiert zu einer übergeben Zahl 1 hinzu, und die Funktion `add` addiert zwei Zahlen. Im Typ einer Funktion trennt ein Pfeil `->` Argument- von Ergebnis-Typ. Die Schreibweise von zweistelligen Funktionen ist gewöhnungsbedürftig. Argumente werden nicht durch Komma getrennt und im Typ erscheinen zwei Pfeile. Was es damit auf sich hat, werden wir später beleuchten. Statt `Int` oder `Float` steht hier als Typ für Zahlen `number`. Das ist eine Typvariable, die sowohl mit `Int` als auch mit `Float` instanziiert werden kann. Die definierten Funktionen können also sowohl mit Ganz- als auch mit Fließkommazahlen aufgerufen werden. Interessant ist, dass Elm in der Lage ist, die Typen der definierten Funktionen automatisch zu inferieren, ohne dass wir Typinformation angeben.

Komplexere Funktionen können wir in einer Datei definieren. Im Online Editor Try Elm können wir Elm Dateien ausführen, ohne Elm zu installieren.

Die `main`-Ausdruck einer Elm Datei beschreibt das Ergebnis ihrer Ausführung. Wir können mit Hilfe des `Html`-Moduls HTML-Seiten als Ergebnis definieren. Das folgende Programm zeigt “Hello World!” im Browser an.

```
import Html

main = Html.text "Hello World!"
```

Das folgende Programm definiert die Funktion `inc` und gibt das Ergebnis eines Beispielaufrufs im Browser aus.

```
import Html

inc : Int -> Int
inc n = n + 1

main = Html.text (toString (inc 41))
```

Diesmal geben wir explizit einen Typ für `inc` an. Da wir `Int` als Argument- und Ergebnistyp verwenden, können wir diese Funktion nicht mit Fließkommazahlen aufrufen.

Das folgende Programm definiert einen Typ-Alias `Person` für Records, die ein `name`- und ein `weight`-Feld haben mit Typen `String bzw.`Float' haben. Diesen Typ-Alias können wir anschließend in Typ-Signaturen für Funktionen verwenden.

```
import Html

type alias Person = { name : String, weight : Float }

hasWeight : Person -> Bool
hasWeight person = person.weight > 0

me : Person
me = { name = "Sebastian", weight = 76.5 }

main = Html.text (toString (hasWeight me))
```

Dieses Programm gibt `True` aus. Wie wir sehen können wir nicht nur Funktionen sondern auch beliebige andere Ausdrücke (wie hier `me`) in Elm Dateien benennen und dann verwenden.

Wir wandeln nun das Programm wie folgt ab:

```
...

greeting : Person -> String
greeting person = if hasWeight person then "Hi!" else "Hi, let’s eat!"

me : Person
me = { name = "Sebastian", weight = 0 }

main = Html.text (greeting me)
```

Diesmal wird im Browser `Hi, let's eat!` angezeigt. Mit Hilfe eines `if-then-else`-Ausdrucks prüft die Funktion `greeting`, ob die übergebene Person etwas wiegt und fordert, falls nicht, zum Essen auf. Anders als in Ruby, werden `if-then-else`-Ausdrücke nicht mit `end` abgeschlossen. Geschachtelte Ausdrücke müssen gegebenenfalls geklammert werden.

Rekursion statt Schleifen

Da es in Funktionalen Programmen weder Anweisungen noch Kontrollstrukturen gibt, werden Schleifen durch Rekursion ausgedrückt. Wir haben bereits früher gesehen, dass Rekursion

gegenüber Schleifen keine Einschränkung der Ausdrucksstärke darstellt, weil alle Schleifen durch Rekursion ausgedrückt werden können. Wir werden später sehen, wie die Modularisierung durch Funktionen höherer Ordung verwendet werden kann, um bestimmte Rekursionsmuster wiederverwendbar zu abstrahieren. Zunächst betrachten wir jedoch noch andere rekursive Funktionen, um uns mit der Elm-Syntax vertraut zu machen.

Die Funktion `prod` berechnet das Produkt aller Elemente einer Liste von Zahlen.

```
import Html

prod list =
  case list of
    [] -> 1
    n :: ns -> n * prod ns

main = Html.text (toString (prod [1,2,3,4,5]))
```

Die Funktion ist mit Hilfe eines `case`-Ausdrucks definiert, mit dem geprüft wird, ob die übergebene Liste leer ist. Die unterschiedlichen Zweige des `case`-Ausdrucks definieren sogenannte Muster und einen Wert. Der Wert des `case`-Ausdrucks selbst ist der Wert des Zweiges, dessen Muster auf das Argument passt. Muster bestehen aus einem Konstruktor des geprüften Wertes und enthalten Variablen anstelle der Konstruktor Argumente. Diese Variablen werden mit den tatsächlichen Argumenten belegt und können im Wert des Zweiges verwendet werden. Das Programm verwendet die `prod`-Funktion, um die Fakultät von 5 im Browser anzuzeigen.

Algebraische Datentypen

Elm erlaubt es, eigene Typen durch Auflistung ihrer Konstruktoren zu definieren. Der folgende Datentyp `Color` stellt die drei Grundfarben dar.

```
type Color = Red | Green | Blue
```

Die Funktion `hexColor` wandelt eine Grundfarbe in eine Hexadezimal-Darstellung des Farbwertes um.

```
hexColor : Color -> String
hexColor color =
  case color of
    Red -> "#FF0000"
    Green -> "#00FF00"
    Blue -> "#0000FF"
```

Wie bei Listen können wir einen `case`-Ausdruck verwenden, um eine Farbe auf ihren Wert zu testen. Der folgende Ausdruck zeigt das Wort “Green” fett und grün im Browser an.

```
main =
  strong
    [style [("color",hexColor Green)]]
    [text (toString Green)]
```

Um die verwendeten Hilfsfunktionen zur HTML-Generierung unqualifiziert benutzen zu können, müssen wir sie entsprechend importieren:

```
import Html exposing ( strong, text )
import Html.Attributes exposing ( style )
```

Konstruktoren können auch Argumente haben. Wenn `Person` ein Datentyp zur Darstellung von Personendaten ist, können wir zum Beispiel den folgenden Datentyp definieren.

```
type User = Anonymous | LoggedIn Person
```

Jetzt ist `Anonymous` ein Wert vom Typ `User`, und für jedes `p` vom Typ `Person` ist `LoggedIn p` ebenfalls vom Typ `User`. Die folgende Funktion liefert die Beschriftung eines Links, der zum An- oder Abmelden verwendet werden kann, je nachdem, was für ein `User` übergeben wird.

```
authLinkText : User -> String
authLinkText user =
  case user of
    Anonymous -> "anmelden"
    LoggedIn person -> person.name ++ " abmelden"
```

Funktionen höherer Ordnung

Funktionen höherer Ordnung sind Funktionen, die andere Funktionen als Argumente oder als Ergebnis haben. Sie sind ein wichtiges Mittel der Funktionalen Programmierung zur Kombination von einfachen Funktionen zu komplexeren. In diesem Abschnitt abstrahieren wir Gemeinsamkeiten von vorher definierten Funktionen und zeigen durch Wiederverwendung abstrahierter Komponenten, wie Funktionen höherer Ordnung zur Modularität von Programmen beitragen.

Die Funktion `prod : List Int -> Int` hatten wir durch einen `case`-Ausdruck mit zwei Zweigen definiert - einen für die leere Liste und einen für nicht-leere. Im Fall einer nicht-leeren

Liste enthält der Zweig einen rekursiven Aufruf mit der Restliste. Dieses Rekursionsschema können wir mit der Funktion `reduce` abstrahieren.[3]

[3] Die Funktion `reduce` ist in Elm als `List.foldr` vordefiniert.

```
reduce f x list =
  case list of
    [] -> x
    y :: ys -> f y (reduce f x ys)
```

Das Ergebnis von `reduce` kann man sich beispielhaft wie folgt verdeutlichen.

```
reduce f x [a,b,c,d,e] = f a (f b (f c (f d (f e x))))
```

Jedes Vorkommen des Cons-Operators wird also durch `f` ersetzt und die abschließende leere Liste durch `x`. Die Funktion `prod` ist eine Instanz dieses Rekursionsschemas, denn es gilt für alle Listen `list` die folgende Gleichung.

```
prod list = reduce (*) 1 list
```

Eine weitere Instanz ist die folgende Funktion `size`[4].

[4] `size` heißt in Elm `List.length`.

```
size : List a -> Int
size list = reduce (\x n -> n + 1) 0 list
```

Die Typvariable `a` steht für einen beliebigen Typ und signalisiert, dass `size` auf Listen mit Elementen beliebigen Typs angewendet werden kann. Der Ausdruck `(\x n -> n + 1)` ist ein **Lambda-Ausdruck**, dessen Wert eine Funktion ist, die (in diesem Fall) zwei Argumente hat, das erste ignoriert und das zweite um eins erhöht. Die oben verwendete Schreibweise `(*)` macht aus dem Multiplikations-Operator eine zweistellige Funktion, die als Lambda-Ausdruck auch als `(\x y -> x * y)` geschrieben werden könnte.

Die gezeigten Gleichungen könnten auch verkürzt wie folgt geschrieben werden.

```
prod = reduce (*) 1
size = reduce (\x n -> n + 1) 0
```

Hier fehlt die Variable `list` für die Argumentliste der Funktionen, und beide Seiten der Gleichungen beschreiben Funktionen. Die Anwendung der dreistelligen Funktion `reduce` auf zwei Argumente liefert also als Ergebnis eine einstellige Funktion, die die noch nicht übergebene Liste als Argument erwartet. Im Allgemeinen liefern solche partiellen Applikationen in Elm als

Ergebnis eine Funktion zurück, die die restlichen Argumente erwartet.

Als weitere Instanzen von `reduce` definieren wir eine Variante der vordefinierten Funktion `++` zum Konkatenieren von Listen.

```
append : List a -> List a -> List a
append xs ys = reduce (::) ys xs
```

Diese Funktion ersetzt also in `xs` jedes Vorkommen des Cons-Operators `(::)` durch `(::)` und die abschließende leere Liste durch `ys`, wodurch im Ergebnis die Listen konkateniert sind. Wir können `append` wiederum verwenden, um eine Liste von Listen zu einer flachen Liste zu konkatenieren.[5]

[5] Die Funktion `flatten` ist in Elm mit dem Namen `List.concat` vordefiniert.

```
flatten : List (List a) -> List a
flatten = reduce append []
```

Hier sind weitere mit Hilfe von `reduce` definierte Funktionen.

```
double : Int -> Int
double n = n + n

doubles : List Int -> List Int
doubles =
  let doubleAndCons n list = double n :: list
  in reduce doubleAndCons []

square : Int -> Int
square n = n * n

squares : List Int -> List Int
squares =
  let squareAndCons n list = square n :: list
  in reduce squareAndCons []
```

Diese Definitionen enthalten weitere Redundanzen, die wir mit Hilfe von Funktionen höherer Ordnung abstrahieren können. Zunächst definieren und verwenden wir zur Verallgemeinerung von `doubleAndCons` und `squareAndCons` eine Funktion `funAndCons`, die eine Funktion als Argument erwartet und eine Funktion als Ergebnis liefert.

```
funAndCons : (a -> b) -> a -> List b -> List b
funAndCons f x xs = f x :: xs

doubles = reduce (funAndCons double) []
squares = reduce (funAndCons square) []
```

Auf der rechten Seite der `funAndCons` definierenden Gleichung können wir den Operator `(::)` auch als zweistellige Funktion schreiben.

```
funAndCons f x xs = (::) (f x) xs
```

Nun können wir das Argument `xs` weglassen und erkennen eine Instanz eines weiteren Musters.

```
funAndCons f x = (:) (f x)
```

`funAndCons` ist die aus der Mathematik bekannte Funktionskomposition aus `f` und `(::)`. Der Operator `>>` zur Funktionskomposition erwartet zwei Funktionen als Argument, liefert eine als Ergebnis und ist in Elm wie folgt definiert.

```
(>>) : (a -> b) -> (b -> c) -> (a -> c)
f >> g = \x -> g (f x)
```

Es gilt also

```
funAndCons f = f >> (::)
```

und wir können `doubles` und `squares` wie folgt vereinfachen.

```
doubles = reduce (double >> (::)) []
squares = reduce (square >> (::)) []
```

Diesen Spezialfall von `reduce`, bei dem eine Funktion auf jedes Element einer Liste angewendet wird, können wir auch mit einer eigenen Funktion benennen.[6]

[6] Die Funktion `collect` ist in Elm mit dem Namen `List.map` vordefiniert.

```
collect : (a -> b) -> List a -> List b
collect f = reduce (f >> (::)) []
```

Diese Funktion ist selbst oft nützlich und kann verwendet werden, um `doubles` und `squares` weiter zu vereinfachen.

```
doubles = collect double
squares = collect square
```

Schließlich definieren wir noch eine Funktion `select`, die diejenigen Elemente aus einer Liste filtert, die ein übergebenes Prädikat `p` erfüllen.[7]

[7] Die Funktion `select` ist in Elm mit dem Namen `List.filter` vordefiniert.

```
select : (a -> Bool) -> List a -> List a
select p = reduce (\x xs -> if p x then x :: xs else xs) []
```

Diese Definition verwendet einen Lambda-Ausdruck dessen Rumpf mit einem `if-then-else`-Ausdruck testet, welche Listenelemente das übergebene Prädikat erfüllen. Die an `select` übergebene Liste wird in der Definition nicht erwähnt, beide Seiten der Gleichung beschreiben also Funktionen, was auch daran erkennbar ist, dass `reduce` partiell auf zwei Argumente angewendet wurde.

Quellen und Lesetipps

- Why Functional Programming Matters

20

Relationale Datenbanken

Datenbanksysteme erlauben Abfrage, Manipulation und Verwaltung gespeicherter Daten. Sie haben um Ziel, Daten dauerhaft, effizient und konsistent zu speichern.

Datenmodelle bestimmen, in welcher Form Daten in der Datenbank angelegt werden. Am häufigsten wird das sogenannte *relationale Datenmodell* verwendet. Dabei werden Daten in Tabellen abgelegt.

Jede Zeile einer Tabelle entspricht einem *Datensatz*. Die Spalten einer Tabelle werden auch *Attribut* genannt und Datensätze bestehen dementsprechend aus *Attributwerten*.

Als Beispiel betrachten wir die folgende Tabelle mit den Attributen **DozentNachname**, **DozentVorname**, und **VorlesungsTitel**.

Tabelle 20.1: Vorlesungsverzeichnis

DozentNachname	DozentVorname	VorlesungsTitel
Huch	Frank	Informatik für Nebenfächler
Fischer	Sebastian	Weiterbildung Informatik
Huch	Frank	Weiterbildung Informatik

Die Tabelle enthält (den Zeilen entsprechend) drei Datensätze mit (den Spalten entsprechend) jeweils drei Attributwerten. Die Attributwerte des ersten Datensatzes sind zum Beispiel die Zeichenketten “Huch”, “Frank”, und “Informatik für Nebenfächler”.

Während die Reihenfolge der Attribute (also Spalten) relevant ist, spielt die Reihenfolge der Datensätze (also Zeilen) im relationalen Datenmodell keine Rolle. (Deshalb ist es fragwürdig, wie eben vom “ersten Datensatz” zu sprechen.) Auch Mehrfachvorkommen von Zeilen werden ignoriert. Eine Tabelle wird also nicht als Liste sondern als *Menge* von Datensätzen interpretiert. Da Datensätze mathematisch als Tupel dargestellt werden können, entspricht eine Tabelle einer Menge von Tupeln, also einer

Relation. So erklärt sich der Name *relationales Datenmodell*.

Schlüssel zum Referenzieren von Datensätzen

Relationale Datenbanken können mehrere Tabellen verwalten, die aufeinander verweisen. Um auf Datensätze einer Tabelle verweisen zu können, müssen diese eindeutig referenzierbar sein. Dies geschieht mit Hilfe sogenannter *Schlüssel*.

Als Beispiel betrachten wir die beiden folgenden Tabellen, die die selben Daten darstellen wie das obige Vorlesungsverzeichnis.

Tabelle 20.2: Vorlesungen

DozentNachname	VorlesungsTitel
Huch	Informatik für Nebenfächler
Fischer	Weiterbildung Informatik
Huch	Weiterbildung Informatik

Tabelle 20.3: Dozenten

DozentNachname	DozentVorname
Huch	Frank
Fischer	Sebastian

Bei dieser Darstellung werden die Vorlesungstitel in der Tabelle **Vorlesungen** nur noch zusammen mit den Nachnamen der Dozenten gespeichert. Die zu den Nachnamen gehörenden Vornamen sind in Tabelle **Dozenten** den Nachnamen zugeordnet. Diese Darstellung setzt vorraus, dass es keine zwei Dozenten mit dem selben Nachnamen gibt. Ansonsten wäre die Zuordnung der Dozenten zu einer Vorlesung nicht eindeutig. Wir gehen zunächst vereinfachend von solcher Eindeutigkeit aus. Das Attribut **DozentNachname** legt dann die Datensätze in der Tabelle **Dozenten** eindeutig fest und wird deshalb *Schlüssel* der Tabelle genannt.

Im Allgemeinen bezeichnet man als *Schlüssel* eine Menge von Attributen, deren Attributwerte die Datensätze einer Tabelle eindeutig festlegen. (Der eben diskutierte Schlüssel ist also eigentlich die einelementige Menge, die nur aus dem Attribut **DozentNachname** besteht.) Eine Tabelle kann mehrere Schlüssel haben. Die Menge aller Attribute einer Tabelle ist immer ein Schlüssel, da Datensätze gemäß des relationalen Datenmodells nicht doppelt vorkommen.

Zum Beispiel ist die Menge **{DozentVorname}** ebenfalls ein

Schlüssel für die Tabelle **Dozenten**, wenn wir vorraussetzen, dass es keine zwei Dozenten mit dem selben Vornamen gibt.

Die Tabelle **Vorlesungen** hat keine einelementigen Schlüssel, da es zu jedem Dozent mehrere Vorlesungen geben kann und umgekehrt. Der einzige Schlüssel der Tabelle **Vorlesungen** ist also die Menge aller Attribute **{DozentNachname,Vorlesungstitel}**. Schlüssel aus mehreren Attributen werden *Verbundschlüssel* genannt.

Benutzer einer relationalen Datenbank müssen zu jeder Tabelle einen Schlüssel angeben, der zur Referenzierung ihrer Datensätze verwendet wird. Dieser so ausgezeichnete Schlüssel einer Tabelle wird *Primärschlüssel* genannt. Weitere Schlüssel können als sogenannte *Sekundärschlüssel* deklariert werden, was hilfreich sein kann, um Suchanfragen zu beschleunigen.

Die Tabelle **Vorlesungen** hat nur einen Schlüssel, der also auch der Primärschlüssel ist. Für die Tabelle **Dozenten** haben wir als Primärschlüssel **{DozentNachname}** gewählt und verwenden entsprechende Attributwerte zur Referenzierung von Dozenten aus der Tabelle **Vorlesungen**. Die Attributmenge **{DozentNachname}** wird deshalb *Fremdschlüssel* in der Tabelle *Vorlesungen* genannt. (Sie ist kein Schlüssel dieser Tabelle!)

Im Allgemeinen müssen Primär- und Fremdschlüssel nicht identisch sein. Es genügt, wenn die zugehörigen Typen der Attribute kompatibel sind.

Die Darstellung mit zwei Tabellen hat gegenüber der ursprünglichen Darstellung als eine einzige Tabelle **Vorlesungsverzeichnis** den Vorteil, dass Vornamen von Dozenten nicht mehr redundant gespeichert werden. Um einen Vornamen eines Dozenten zu ändern, braucht nur noch ein einziger Datensatz in der Tabelle **Dozenten** geändert zu werden statt aller zugehörigen Datensätze in der Tabelle **Vorlesungsverzeichnis**. Dadurch wird auch vermieden, dass der selbe Dozent versehentlich mit verschiedenen Vornamen gespeichert wird.

Unsere Annahme, dass Dozenten eindeutig über ihren Nachnamen (oder Vornamen) identifiziert werden können ist unrealistisch. Statt wie in der Tabelle **Vorlesungen** die Menge aller Attributwerte als Primäschlüssel zu verwenden, können wir der Tabelle künstlich ein Attribut hinzufügen, welches die Datensätze eindeutig festlegt. Relationale Datenbanksysteme unterstützen die Definition solcher Attribute mit einer fortlaufenden Nummer als Attributwert. Durch die fortlaufende Nummerierung sind solche Attribute automatisch Schlüssel und werden *Surrogatschlüssel* genannt.

Um Dozenten mit gleichen Vor- oder Nachnamen nicht von vornherein auszuschließen, ändern wir die Tabelle **Dozenten** wie folgt.

Tabelle 20.4: Dozenten

DozentID	DozentNachname	DozentVorname
0	Huch	Frank
1	Fischer	Sebastian

Auch ohne die Eindeutigkeit von Vor- oder Nachnamen vorauszusetzen ist nun **{DozentID}** ein Schlüssel der Tabelle **Dozenten**. Wenn wir ihn als Primärschlüssel festlegen, können wir statt **{DozentNachname}** nun **{DozentID}** als Fremdschlüssel in der Tabelle **Vorlesungen** verwenden.

Tabelle 20.5: Vorlesungen

DozentID	VorlesungsTitel
0	Informatik für Nebenfächler
1	Weiterbildung Informatik
0	Weiterbildung Informatik

Diese Darstellung eines Vorlesungsverzeichnis enthält noch immer Redundanz, da die Vorlesungstitel mehrfach gespeichert werden. Um auch diese Redundanz zu eliminieren, zerlegen wir die Tabelle **Vorlesungen** wie folgt in zwei Tabellen.

Tabelle 20.6: IstDozent

DozentID	VorlesungsID
0	0
1	1
0	1

Tabelle 20.7: Vorlesungen

VorlesungsID	VorlesungsTitel
0	Informatik für Nebenfächler
1	Weiterbildung Informatik

Zur Verknüpfung der Dozenten mit Vorlesungen verwenden wir nun die Tabelle **IstDozent**. Diese referenziert die Tabellen **Dozenten** und **Vorlesungen** jeweils über Fremdschlüssel. Die Tabelle **Vorlesungen** identifiziert die Vorlesungstitel mit Hilfe des Surrogatschlüssels **{VorlesungsID}**, der auch ihr Primärschlüssel ist.

Die Tabelle **IstDozent** ist die einzige ohne einen einelementigen Schlüssel. Sie representiert eine sogenannte

N-zu-M-Beziehung zwischen Dozenten und Vorlesungen. Neben N-zu-M-Beziehungen, die mit einer extra Tabelle dargestellt werden, gibt es auch 1-zu-1- und 1-zu-N-Beziehungen. Diese können einfacher (und ohne unnötige Redundanz) ohne extra Tabelle dargestellt werden. Hätte zum Beispiel jede Vorlesung nur einen Dozenten, so würde es genügen, ein zusätzliches Attribut **DozentID** als Fremdschlüssel in der Tabelle **Vorlesungen** zu verwenden. In diesem Fall wäre also die Version 20.5 der Tabelle **Vorlesungen** bereits redundanzfrei.

Ein Sonderfall ergibt sich bei 1-zu-1- und 1-zu-N-Beziehungen, wenn statt der 1 auch eine 0 zugelassen sein soll. Gemäß Version 20.5 der Tabelle **Vorlesungen** muss es zu jeder Vorlesung einen Dozenten geben. Ein Vorlesungstitel ohne Dozent lässt sich in die Tabelle nicht eintragen, ohne den Wert des Attributs **DozentID** leer zu lassen. Um dies zu erlauben, gibt es den sogenannten **NULL**-Wert, der als undefinierter Attributwert fungiert.

Die drei Tabellen **Dozenten**, **IstDozent** und **Vorlesungen** representieren die selbe Information wie die ursprüngliche Tabelle **Vorlesungsverzeichnis**, wobei Redundanz in der ursprünglichen Darstellung eliminiert wurde. Die ursprüngliche Tabelle kann in gängigen relationalen Datenbanksystemen als sogenannte *Sicht* extrahiert werden.

Daten-Integrität

Relationale Datenbanksysteme implementieren verschiedene Konsitenzprüfungen, die sicherstellen, dass die gespeicherten Daten sinnvoll interpretiert werden können.

Die einfachste Integritäts-Bedingung ist die sogenannte *Bereichsintegrität*. Diese fordert, dass Attributwerte zu einem dem Attribut zugeordneten Wertebereich (bzw. Typ) gehören. Zum Beispiel müssen im obigen Beispiel die Werte der Attribute **DozentID** und **VorlesungsID** Zahlen sein. Das Einfügen von Datensätzen mit ungültigen Attributwerten wird vom Datenbanksystem verhindert.

Die Forderung, dass der Primärschlüssel einer Tabelle die enthaltenen Datensätze eindeutig festlegt, wird als *Entitätsintegrität* bezeichnet. Das Einfügen von Datensätzen, deren zum Primärschlüssel gehörige Werte bereits in einem anderen Datensatz vorkommen, wird vom Datenbanksystem verhindert.

Aus der Referenzierung von Tabellen untereinander ergibt sich der Begriff der *referentiellen Integrität*. Diese fordert, dass zu den Werten eines Fremdschlüssels ein entsprechender Datensatz in der referenzierten Tabelle existiert. Gegebenenfalls ist auch **NULL** als Fremdschlüsselwert erlaubt (siehe oben).

Verboten sind jedoch in jedem Fall Werte ungleich **NULL** zu denen kein Datensatz existiert. Ein Datenbanksystem verhindert das Einfügen eines Datensatzes, deren Fremdschlüssel keinen existierenden Datensatz referenziert.

Referentielle Integrität kann nicht nur durch Einfügen in der referenzierenden Tabelle sondern auch durch Löschen (oder Ändern) von Datensätzen in der referenzierten Tabelle verletzt werden. Um dies zu verhindern, gibt es verschiedene Strategien.

Die einfachste Strategie ist, das Löschen von referenzierten Datensätzen zu verbieten.

Falls **NULL**-Werte als Fremdschlüssel erlaubt sind, können Referenzen auf einen gelöschten Datensatz durch **NULL** überschrieben und dadurch gelöscht werden.

Eine dritte Strategie ist sogenannte Löschweitergabe, bei der referenzierende Datensätze zusammen mit dem referenzierten Datensatz gelöscht werden. Diese Vorgehensweise bietet sich insbesondere bei Tabellen an, die nur aus Fremdschlüsseln bestehen (wie **IstDozent** im obigen Beispiel), da dabei nur Referenzen gelöscht werden.

Neben den diskutierten Konsistenzbedingungen wird auch sogenannte *logische Integrität* betrachtet, die aber in der Regel nicht vom Datenbanksystem unterstützt wird. Zum Beispiel könnte man fordern, dass eine Vorlesung nur eine bestimmte Anzahl von Dozenten haben darf. Solche Bedingungen müssen von Anwendern eines Datenbanksystems selbst erfüllt werden.

Datenbankprogrammierung mit Ruby

In Ruby können wir mit dem `sqlite3` Gem auf SQLite-Datenbanken zugreifen. Nach der Installation des Pakets, können wir zu Beginn eines Programms `require 'sqlite3'` schreiben, um das Paket zu verwenden.

Das folgende Programm öffnet die zuvor angelegte Datenbank und gibt testweise alle Einträge der Tabelle Vorlesungen aus.

```
require 'sqlite3'

db = SQLite3::Database.new "IQSH.sqlite"
rows = db.execute "SELECT * FROM Vorlesungen"
p rows
```

Die Ausgabe dieses Programms hängt davon ab, welche Einträge in der Datenbank gespeichert sind. Sie könnte zum Beispiel wie folgt lauten.

```
[[1, "Informatik für Nebenfächler"], [2, "IQSH Weiterbildung"]]
```

Das Ergebnis des Aufrufs von `db.execute` ist also ein Array von Zeilen, die wiederum als Array von Einträgen dargestellt sind. Um die Zeilen einer Tabelle untereinander auszugeben, definieren wir die folgende Prozedur.

```
def print_rows table
  table.each do |row|
    puts row.join("|")
  end
end
```

Wenn wir diese im obigen für die Ausgabe verwenden, ändert sie sich wie folgt.

```
1|Informatik für Nebenfächler
2|IQSH Weiterbildung
```

Die Einträge jeder Zeile werden also durch senkrechte Striche getrennt ausgegeben. Durch den folgenden Aufruf können Zeilen auch als Hash statt als Array dargestellt werden.

```
db.results_as_hash = true
```

Anschließend liefert `db.execute` ein Array von Hashes zurück, in der auf Einträge sowohl über Spaltennamen als auch über Positionen zugegriffen werden kann. Die obige Anfrage liefert zum Beispiel das folgende Ergebnis.

```
[{ "Id" => 1, "Titel" => "Informatik für Nebenfächler"
 , 0 => 1, 1 => "Informatik für Nebenfächler"
 }
,{ "Id" => 2, "Titel" => "IQSH Weiterbildung"
 , 0 => 2, 1 => "IQSH Weiterbildung"
 }
]
```

Falls im Ergebnis einer SQL-Anfrage Spaltennamen doppelt verwendet werden, kann nicht auf alle Einträge mit Hilfe von Spaltennamen zugegriffen werden. Durch den Zugriff über Positionen sind aber alle Einträge selbst eines solchen Ergebnisses erreichbar.

Die `execute`-Methode auf Datenbanken hat einen optionalen zweiten Parameter, der verwendet werden kann, um Argumente an SQL-Anfragen zu übergeben. Dazu können in SQL-Anfragen Fragezeichen als Wildcard verwendet werden, die von links nach rechts durch Elemente eines zusätzlich übergebenen Arrays ersetzt werden. Auf diese Weise ist sichergestellt, dass die Argumente korrekt in SQL dargestellt werden, was auch sogenannten SQL-Injection Angriffen vorbeugt. Das folgende Programm veranschaulicht den zusätzlichen Parameter.

```
dozent = "Frank"
result = db.execute """
SELECT Titel FROM VorlesungsVerzeichnis WHERE Vorname = ?;
""", [dozent]
result.each do |row|
  puts row["Titel"]
end
```

Im folgenden erzeugen wir eine Filmdatenbank allein mit Hilfe von SQL-Anweisungen über die Ruby-Schnittstelle. Die dazu nötigen Tabellen legen wir mit den folgenden `CREATE TABLE` Anweisungen an.

```
movie_db = SQLite3::Database.new "movies.sqlite"
movie_db.results_as_hash = true

movie_db.execute """
CREATE TABLE IF NOT EXISTS person (
  id INTEGER PRIMARY KEY AUTOINCREMENT NOT NULL UNIQUE,
  name TEXT NOT NULL);
"""

movie_db.execute """
CREATE TABLE IF NOT EXISTS movie (
  id INTEGER PRIMARY KEY AUTOINCREMENT NOT NULL UNIQUE,
  title TEXT NOT NULL,
  year INTEGER NOT NULL,
  director INTEGER NOT NULL);
"""

movie_db.execute """
CREATE TABLE IF NOT EXISTS actor (
  id INTEGER PRIMARY KEY AUTOINCREMENT NOT NULL UNIQUE,
  person INTEGER NOT NULL,
  movie INTEGER NOT NULL);
"""
```

Wir wollen nun die wie folgt als Hash dargestellten Filme in die Datenbank eintragen.

```
movies = [
  { "title" => "Zwei glorreiche Halunken", "year" => 1966,
    "director" => "Sergio Leone",
    "actors" => ["Clint Eastwood", "Eli Wallach"] },
  { "title" => "Fight Club", "year" => 1999,
    "director" => "David Fincher",
    "actors" => ["Brad Pitt", "Edward Norton"] },
  { "title" => "Gran Torino", "year" => 2008,
    "director" => "Clint Eastwood",
```

```
    "actors" => ["Clint Eastwood", "Bee Vang", "Christopher Carley"] }
]
```

Dazu verwenden wir drei Hilfsfunktionen zum Einfügen von Werten in die angelegten Tabellen. Diese Hilfsfunktionen fügen den übergebenen Eintrag in die entsprechende Tabelle ein, falls er noch nicht existiert. In jedem Fall wird die `id` des einzufügenden Eintrag zurück geliefert. Wenn der Eintrag bereits existiert, wird die existierende `id` geliefert.

```
def insert_person db, name
  result = db.execute """
  SELECT id FROM person WHERE name = ?;
  """, [name]

  if result.size > 0 then
    return result.first["id"]
  end

  db.execute """
  INSERT INTO person (name) VALUES (?);
  """, [name]

  return insert_person db, name
end

def insert_movie db, movie
  result = db.execute """
  SELECT id FROM movie WHERE title = ? AND year = ? AND director = ?;
  """, [movie["title"], movie["year"], movie["director"]]

  if result.size > 0 then
    return result.first["id"]
  end

  db.execute """
  INSERT INTO movie (title, year, director) VALUES (?,?,?);
  """, [movie["title"], movie["year"], movie["director"]]

  return insert_movie db, movie
end

def insert_actor db, actor
  result = db.execute """
  SELECT id FROM actor WHERE movie = ? AND person = ?;
  """, [actor["movie"], actor["person"]]

  if result.size > 0 then
    return result.first["id"]
```

```
  end

  db.execute """
  INSERT INTO actor (movie, person) VALUES (?,?);
  """, [actor["movie"], actor["person"]]

  return insert_actor db, actor
end
```

Diese Hilfsfunktionen können wir verwenden, um die gezeigten Filme in die Datenbank einzutragen.

```
movies.each do |movie|
  movie["director"] = insert_person movie_db, movie["director"]
  movie_id = insert_movie movie_db, movie
  movie["actors"].each do |actor|
    actor_id = insert_person movie_db, actor
    insert_actor movie_db, { "movie" => movie_id, "person" => actor_id }
  end
end
```

Bei genauerer Betrachtung fällt auf, dass die Implementierungen der Hilfsfunktionen zum Einfügen von Datensätzen sich ähneln. Wir werden die bisherigen Definitionen im Folgenden schrittweise so abstrahieren, dass wiederverwendbare Fragmente entstehen, die es erlauben, unser Programm kompakter zu implementieren.

Zunächst verallgemeinern wir die `insert_`-Funktionen zu einer einzigen Funktion `insert`, mit zwei zusätzlichen Parametern für die Namen der Tabelle und ihrer Spalten.

```
def insert db, table_name, col_names, values
  condition = col_names.collect { |col| col + " = ?"}.join(" AND ")
  result = db.execute """
  SELECT id FROM #{table_name} WHERE #{condition};
  """, col_names.collect { |col| values[col] }

  if result.size > 0 then
    return result.first["id"]
  end

  db.execute """
  INSERT INTO #{table_name} (#{col_names.join(',')})
  VALUES (#{(['?'] * col_names.size).join(',')});
  """, col_names.collect { |col| values[col] }

  return insert db, table_name, col_names, values
end
```

Statt unterschiedliche Hilfsfunktionen für die unterschiedlichen Tabellen aufzurufen, können wir nun für alle Tabellen die Funktion `insert` verwenden, um Einträge einzufügen, indem wir zusätzlich die Namen der Tabelle und ihrer Spalten übergeben.

Im nächsten Schritt definieren wir eine Klasse für Tabellen, die die übergebenen Namen als Attribute speichert und eine Methode `insert` bereitstellt, die nur noch die einzufügenden Werte als Argument erwartet.

```
class Table
  def initialize(db, name, cols)
    @db = db
    @name = name
    @cols = cols
  end

  def insert values
    condition = @cols.collect { |col| col + " = ?"}.join(" AND ")
    result = @db.execute """
    SELECT id FROM #{@name} WHERE #{condition};
    """, @cols.collect { |col| values[col] }

    if result.size > 0 then
      return result.first["id"]
    end

    @db.execute """
    INSERT INTO #{@name} (#{@cols.join(',')})
    VALUES (#{(['?'] * @cols.size).join(',')});
    """, @cols.collect { |col| values[col] }

    return insert values
  end
end
```

Statt `Table`-Objekte von Hand zu erzeugen, definieren wir eine Klasse `Database` mit einer Methode, die die Spaltennamen anhand des Tabellennamens aus der Datenbank extrahiert und ein `Table`-Objekt zurück liefert.

```
class Database
  def initialize(file_name)
    @db = SQLite3::Database.new file_name
    @db.results_as_hash = true
  end

  def execute sql, args = nil
```

```
    begin
      return @db.execute sql, args
    rescue SQLite3::SQLException => e
      puts e.message
      return nil
    end
  end

  def [](table_name)
    result = execute """
    PRAGMA table_info(#{table_name})
    """

    if result.empty? then
      return nil
    else
      cols = result.collect do |row|
        row["name"]
      end.select do |col|
        col != "id"
      end

      return Table.new self, table_name, cols
    end
  end
end
```

Der Konstruktor erzeugt ein Datanbank-Objekt und speichert es in der Attribut-Variablen `@db`. Die Methode `execute` ruft die entsprechende Methode des gespeicherten Datenbank-Objektes auf und fügt Fehlerbehandlung hinzu. Die Methode `[]` schließlich, fragt mit Hilfe einer `PRAGMA`-Anfrage die Spaltennamen der übergebenen Tabelle ab und liefert ein entsprechendes `Table`-Objekt zurück. Falls in der Datenbank keine Tabelle mit dem übergebenen Namen existiert, wird stattdessen `nil` zurück geliefert.

Als letzten Schritt fügen wir der Klasse `Database` eine Methode zum Anlegen von Tabellen hinzu.

```
class Database
  def []=(table_name, col_defs)
    execute """
    DROP TABLE IF EXISTS #{table_name};
    """
    execute """
    CREATE TABLE #{table_name} (#{col_defs});
    """
    return nil
```

```
  end
end
```

Hier wird eine ggf. existierende gleichnamige Tabelle aus der Datenbank entfernt, bevor eine neue Tabelle erzeugt wird. Auf diese Weise ist sicher gestellt, dass nach dem Aufruf eine leere Tabelle in der Datenbank existiert, deren Struktur der gegebenen Spaltenbeschreibung entspricht.

Wenn wir die gezeigten Klassen in einer Datei `database_sqlite.rb` speichern, können wir sie wie folgt zur Implementierung des Programms, das unsere Filmdatenbank erzeugt, verwenden.

```
require_relative "database_sqlite"

movie_db = Database.new "movies.sqlite"

if movie_db["person"] == nil then
  movie_db["person"] = """
    id INTEGER PRIMARY KEY AUTOINCREMENT NOT NULL UNIQUE,
    name TEXT NOT NULL
  """
end

if movie_db["movie"] == nil then
  movie_db["movie"] = """
    id INTEGER PRIMARY KEY AUTOINCREMENT NOT NULL UNIQUE,
    title TEXT NOT NULL,
    year INTEGER NOT NULL,
    director INTEGER NOT NULL
  """
end

if movie_db["actor"] == nil then
  movie_db["actor"] = """
    id INTEGER PRIMARY KEY AUTOINCREMENT NOT NULL UNIQUE,
    person INTEGER NOT NULL,
    movie INTEGER NOT NULL
  """
end

movies = [
  { "title" => "Zwei glorreiche Halunken", "year" => 1966,
    "director" => "Sergio Leone",
    "actors" => ["Clint Eastwood", "Eli Wallach"] },
  { "title" => "Fight Club", "year" => 1999,
    "director" => "David Fincher",
    "actors" => ["Brad Pitt", "Edward Norton"] },
  { "title" => "Gran Torino", "year" => 2008,
    "director" => "Clint Eastwood",
```

```
    "actors" => ["Clint Eastwood", "Bee Vang", "Christopher Carley"] }
]

movies.each do |movie|
  movie["director"] = movie_db["person"].insert({
    "name" => movie["director"]
  })
  movie_id = movie_db["movie"].insert movie
  movie["actors"].each do |actor|
    actor_id = movie_db["person"].insert({
      "name" => actor
    })
    movie_db["actor"].insert({
      "movie" => movie_id,
      "person" => actor_id
    })
  end
end
```

21 Netzwerke

Bisher haben wir Computer als isolierte Einheit betrachtet, die mit Hilfe von Schaltwerken von Nutzern geschriebene Programme ausführen. Wegen der hohen Kosten von Computern entstand der Wunsch, diese gemeinsam, auch über große Entfernungen, zu nutzen. Dazu wurde es notwendig, Computer zu vernetzen.

Heute gibt es kaum noch Computer, die nicht ans Internet angeschlossen sind. Selbst Mobiltelefone sind oft pausenlos online. Im **Internet** werden unterschiedliche **Dienste** (Webseiten, Dateitransfer, Email, usw.) über unterschiedliche **Transportmedien** (lokale Netze, Drahtlosnetzwerke, Mobilfunknetze, usw.) bereitgestellt.

In diesem Abschnitt interessieren wir uns für die **Abstraktionsmechanismen**, die es ermöglichen, eine solche Vielfalt von Medien und Diensten beherrschbar zu kombinieren. Wir werden sehen, dass die zu Grunde liegenden Ideen, denen ähneln, die dazu dienen, Programme unterschiedlicher Programmiersprachen auf unterschiedlicher Hardware auszuführen.

Es gibt unterschiedliche Kriterien, Computernetze zu klassifizieren. Bei der Klassifizierung nach Entfernung der beteiligten Rechner wird unterschieden zwischen Local Area Networks (**LAN**, im selben Gebäude), Metropolitain Area Networks (**MAN**, über wenige 100 Kilometer) und Wide Area Networks (**WAN**, bis zu weltweit).

Auch nach der Art der Vernetzung, der sogenannten **Topologie**, können wir unterscheiden.

- In **Maschen**-Netzen ist jeder Rechner mit jedem anderen direkt verbunden.
- In einem **Bus**-Netz sind alle Rechner mit einem gemeinsamen Kommunikationsmedium verbunden.
- In einem **Stern**-Netz sind alle Rechner mit einer zentralen Einheit verbunden, die die Kommunikation steuert.

- In einem **Ring**-Netz sind Rechner ringförmig verbunden und Daten werden reihum weiter geleitet.
- In einem **Baum**-Netz sind Rechner hierarchisch miteinander verbunden.

Maschen-Netze sind nur für sehr kleine Anzahlen von Computern praktikabel. Selbst in lokalen Netzen werden in der Regel Bus-Netze verwendet. Stern-Netze eignen sich für größere lokale Netzwerke oder Firmennetze und erlauben es, das Netzwerk an einem zentralen Punkt zu administrieren. Ring-Netze sind historisch gesehen eine kostengünstige Alternative, um mehrere Standorte einer Firma zu verbinden. Heutzutage geschieht das in der Regel über das Internet, das Baum-artige Strukturen mit freieren Formen der Vernetzung kombiniert.

Dei der Netzwerkkommunikation wird unterschieden zwischen dem Senden einer Nachricht zu genau einem Empfänger (**unicast**), zu mehreren ausgewählten Empfängern (**multicast**) und zu allen Netzwerkteilnehmern gleichzeitig (**broadcast**).

Netzwerkdienste

Die Post transportiert Sendungen von Absendern zu Empfängern. Dabei gibt es festgelegt Formate für Briefe, Päckchen und Pakete. Auch ist festgelegt, wie und wo Absender und Empfängeradressen zu notieren sind. Der Standardversand garantiert nicht, bis wann oder dass Sendungen überhaupt ankommen. Per Expressversand und Einschreiben kann die Zustellung beschleunigt und garantiert werden.

In Computernetzen verhält es sich ganz ähnlich. Auch hier werden **Formate** für den Austausch von Daten definiert, die **Adressierung** folgt festgelegten Regeln und unterschiedliche Netzwerdienste unterscheiden sich bezüglich der zugesicherten **Dienstgüte**. Ein interessanter Aspekt ist dabei die automatische **Fehlererkennung** und -Korrektur zur Bereitstellung verlässlicher Dienste.

In Computernetzen wird zwischen **verbindungslosen** und **verbindungs-orientierten** Diensten unterschieden. Verbindungslose Dienste funktionieren ähnlich wie die Post. Sie unterscheiden sich bezüglich ihrer **Verlässlichkeit**, also danach, ob Nachrichten

- verloren gehen (oder dupliziert),
- während des Transprts verfälscht,
- oder in ihrer Reihenfolge vertauscht

werden können.

Verbindungs-orientierte Dienste funktionieren ähnlich wie das Telefon. Dem sogenannten **Verbindungsaufbau** folgt der **Datenaustausch** vor dem **Verbindungsabbruch**. Die Kommunikation während des Datenaustauschs erfolgt entweder nur ein eine Richtung (**simplex**) oder in beide Richtungen (**duplex**).

Das Internet stellt einen verbindungslosen Kommunikationsdienst bereit. Es gibt jedoch Anwendungen, die auf Basis des Internets verbindungs-orientierte Dienste bereitstellen (z.B. Internettelefonie).

Netzwerkprotokolle

Die Regeln, nach denen die Kommunikation in Netzwerken abläuft, werden in Protokollen definiert. Protokolle spezifizieren zum Beispiel Datenformate und Adressierungsschemata sowie Mechanismen zur Weiterleitung von Nachrichten oder zur Fehlerkorrektur. Auch Mechanismen zum Aushandeln von Übertragungsparametern oder zum Verbindungsauf- und Abbau sind Teil von Protokollen.

Um verschiedene Aspekte der Kommunikation zu entkoppeln sind Netzwerkprotokolle hierarchisch in sogenannten **Schichten** strukturiert, die aufeinander aufbauen. Dieses Vorgehen ähnelt der Ausführung von Programmen auf einem Rechner, wo Programme zunächst in Assemblersprache übersetzt werden, und die Assemblerprogramme schließlich in Maschinensprache. Auch die Abstraktion von Algorithmen durch Funktionen und Prozeduren folgt insofern einem ähnlichen Muster, als zur Verwendung einer Funktion oder Prozedur ihre Implementierung nicht bekannt zu sein braucht.

Unterschiedliche Netzwerkprotokolle ordnen sich in der Regel in eine der folgenden Schichten ein, je nachdem welche Funktionalität der Kommunikation sie implementieren.

- Die **Verbindungsschicht** umfasst Protokolle zur Übertragung von Bitfolgen über ein physikalisches Transportmedium. Die übertragenen Daten als Bitfolgen zu interpretieren ist bereits eine Abstraktion des eigentlichen über das Transportmedium verschickten Signals.[1] Ein wichtiger Aspekt von Protokollen dieser Schicht ist die Erkennung und Korrektur von Übertragungsfehlern. Häufig eingesetzte Protokolle in dieser Schicht sind Ethernet oder WLAN.
- Die **Vernetzungsschicht** umfasst Protokolle zur Weiterleitung von Nachrichten durch räumlich getrennte Netzwerke. Alle beteiligten Netze müssen dazu natürlich physikalisch verbunden sein. Es ist jedoch möglich, eine Nachricht über

[1] Bits (also Nullen und Einsen) als elektrisches Signal in Kupferkabeln, optisches Signal in Glasfaserkabeln oder Funk-Signal zur Drahtlosübertragung zu konvertieren, ist nicht Teil der Informatik sondern (ähnlich wie die physikalische Realisierung von Schaltwerken) Teil der Elektro- bzw. Nachrichtentechnik.

unterschiedliche Transportmedien weiter zu leiten. Ein wichiger Aspekt von Protokollen dieser Schicht sind Mechanismen zur Etablierung eines Weges vom Sender zum Empfänger über verschiedene physikalische Netzwerke hinweg. Im Internet wird diese Schicht von dem Internetprotokoll (IP) implementiert.

- Die **Transportschicht** umfasst Protokolle zur Etablierung gewisser Gütekriterien für die Übertragung von Nachrichten. Insbesondere werden Übertragungsfehler der Vernetzungsschicht durch Mechanismen kompensiert, die die verlässliche Übertragung von Nachrichten gewährleisten. Im Internet wird diese Schicht in der Regel vom Transmission Control Protocol (TCP) implementiert.

- Die **Anwedungsschicht** umfasst Protokolle zum anwendungsspezifischen Nachrichtenaustausch. Ein im Internet häufig eingesetztes Protokoll dieser Schicht ist das HyperText Transfer Protocol (HTTP) zur Übertragung von Webseiten über das Internet. Auch Protokolle wie SMTP, POP oder IMAP zur Übertragung von Emails, FTP zum Dateitransfer oder Protokolle zur Internettelefonie gehören in diese Schicht.

Die Unterteilung der Protokolle in die vier genannten Schichten ist ein wichtiger Mechanismus zur **Abstraktion** der bereitgestellten Dienste. So brauchen sich Protokolle der Anwendungsschicht nicht darum zu kümmern, über welches Medium Daten transportiert werden oder wie sie vom Sender zum Empfänger gelangen. Umgekehrt ist es den Protokollen der Verbindungsschicht egal, welche Art von Daten sie über ein Transportmedium transportieren, ob es sich also zum Beispiel um Webseiten oder Videodaten handelt.

Das Internet umfasst Protokolle von der Vernetzungs- bis zur Anwendungsschicht, die wir exemplarisch in folgenden Abschnitten behandeln werden. Zunächst schauen wir uns jedoch noch wichtige Aspekte der Verbindungsschicht an.

Media Access Control und Fehlererkennung

Lokale Netzwerke sind in der Regel Bus-Netze, in denen also alle Rechner mit einem gemeinsamen Transportmedium verbunden sind. Im Ethernet sind Computer mit zusammengeschalteten Kabeln verbunden, im WLAN teilen sie sich einen gemeinsamen Funkkanal.

Bei gemeinsamer Nutzung eines Transportmediums können sogenannte **Kollisionen** auftreten, wenn mehrere Parteien gleichzeitig versuchen, Daten zu senden. Mechanismen zur

Media Access Control (**MAC**) dienen dazu, Kollisionen zu behandeln. Sogenannte **pessimistische** Verfahren versuchen dabei Kollisionen von vornherein zu vermeiden, während **optimistische** Verfahren versuchen, geeignet auf entstandene Kollisionen zu reagieren.

Im Ethernet oder WLAN werden Computer über eine sogenannte MAC-Adresse eindeutig identifiziert. Das Format zum Austausch von Daten im Ethernet besteht im Wesentlichen aus einem Header, der die Absender- und Empfängeradresse enthält, gefolgt den eigentlichen Nutzdaten und einer Prüfsumme zur Fehlerbehandlung.

Eine Prüfsumme ist eine Methode, um Übertragungsfehler durch Redundanz zu erkennen. Als vereinfachtes Beispiel dieses Prinzips könnten wir jeder Nachricht ein Bit anhängen, dass angibt, ob die Nutzdaten eine gerade oder eine ungerade Anzahl von Einsen enthalten. Beim Dekodieren der Nachricht können (manche) Übertragungsfehler dann daran erkannt werden, dass das angehängte Bit nicht zu der empfangenen Anzahl von Einsen passt.

Zum Beispiel könnten wir die Nachricht

```
011011
```

durch eine angehängte `0` ergänzen, die anzeigt, dass sie eine gerade Anzahl von Einsen enthält:

```
0110110
```

Dadurch ist sicher gestellt, das gesendete Nachrichten immer eine gerade Anzahl von Einsen enthalten. Denn wenn die Anzahl ursprünglich ungerade ist, wird ja eine zusätzliche Eins angehängt. Angenommen, die obige Nachricht würde bei der Übertragung wie folgt verfälscht:

```
0100110
```

Der Empfänger würde dann einen Übertragungsfehler daran erkennen, dass die Anzahl der Einsen in der empfangenen Nachricht ungerade ist. anhand dieser Information kann der Empfänger den Fehler zwar erkennen aber nicht korrigieren, da nicht klar ist, an welcher Stelle der Fehler auftrat.

Auch werden nicht alle Übertragungsfehler auf diese Weise erkannt. Wenn zum Beispiel mehr als ein Bit verfälscht wird, kann es passieren, dass der Fehler nicht erkannt wird (zum Beispiel, wenn genau zwei Einsen jeweils durch Nullen ersetzt werden oder umgekehrt). Um mehr Fehler zu erkennen, kann die Prüfsumme verlängert werden. Zum Beispiel könnten wir zwei Bits anhängen, die dem Rest bei der Division durch vier entsprechen. Die Nachricht

```
010100
```

würde also wie folgt verlängert, da “10” die Binärdarstellung des Restes der Division von zwei durch vier ist.

```
01010010
```

Diese Methode erlaubt es mehr, wenn auch noch immer nicht alle, Übertragungsfehler zu erkennen. Zum Beispiel würden wir erkennen, wenn zwei Nullen in der ursprünglichen Nachricht durch Einsen ersetzt würden, da dann die Anzahl zwar noch immer durch zwei aber nun auch durch vier teilbar wäre.

Internet

Dieser Abschnitt liefert einen Überblick über Mechanismen und Protokolle der Vernetzungs-, Transport- und Anwendungsschicht, wie sie im Internet verwendet werden.

Die Vernetzungsschicht wird im Internet vom Internet Protocol (IP) implementiert. Die Aufgabe dieses Protokolls ist es, Datenaustausch über mehrere sogenannte Router zu ermöglichen, und wir werden die Grundideen des Routings sowie das zugrunde liegende Adressierungsschema der IP-Adressen skizzieren.

Auf der Transportschicht kommen im Internet das verbindungslose unzuverlässige User Datagram Protocol (UDP) und das verbindungsorientierte zuverlässige Transmission Control Protocol (TCP) zum Einsatz. Uns interessieren vor allem TCP und die Mechanismen, die zuverlässige Kommunikation über ein unzuverlässiges Netzwerk ermöglichen.

Schließlich skizzieren wir das Domain Name System (DNS) zur Namensauflösung im Internet, das Format und Mechanismen zum Austausch von Email, sowie das HyperText Transfer Protocol (HTTP) als Grundlage des World Wide Web (WWW).

Routing und IP-Adressen

Die Vernetzungsschicht implementiert im Internet einen unzuverlässigen Datenaustauschdienst auf Basis der Verbindungsschicht. Sie abstrahiert von dem Übertragungsverfahren und erlaubt es so, Daten unabhängig von der Netzwerktechnologie zu übertragen.

Im Internet werden Daten über sogenannte Router vom Absender zum Empfänger weiter geleitet. Damit das funktioniert, muss jeder Router im Internet wissen, über welchen seiner

direkten Nachbarn welche anderen Rechner im Internet erreichbar sind. Dazu speichern Router eine sogenannte Routing-Tabelle, die eine Zuordnung von IP-Adressen oder Adressbereichen zu Netzwerkschnittstellen speichert, über die Daten an die entsprechende Adresse weitergeleitet werden sollen.

In Version 4 des Internet Protokolls bestehen Adressen aus 32 Bit, von denen eine variable Anzahl Bits das Subnetzwerk und die restlichen Bits einen Rechner[2] in diesem Subnetzwerk spezifizieren. Die Subnetze sind im Internet regional hierarchisch angeordnet, so dass Router in einer Region der Welt nicht für jeden Rechner in einer anderen Region einen Eintrag in der Routing-Tabelle abspeichern müssen sondern nur einen Eintrag für alle Rechner im entsprechenden Subnetz. IP-Adressen werden als durch Punkte getrennte Dezimalzahlen notiert. Der Webserver der Uni-Kiel hat zum Beispiel die Adresse `134.245.13.21`.

[2] In Wirklichkeit wird durch die IP-Adresse nicht ein Rechner sonder nein sogenannter Host spezifiziert, der mit seiner Netzwerkschnittstelle identifiziert wird.

Es gibt verschiedene Verfahren, Routing-Tabellen in Routern eines Netzwerkes zu verwalten.

In Netzwerken aus wenigen Rechnern können Routing-Tabellen von einem Administrator festgelegt und auf allen Routern verteilt werden. Dieses sogenannte statische Routing ist jedoch unflexibel, da bei Änderungen der Netzwerktopologie (zum Beispiel durch Hinzufügen oder Ausfall eines Routers) die Routing-Tabellen auf allen Routern von Hand aktualisiert werden müssen.

Ein verteilter Mechanismus zur Verwaltung der Routing-Tabellen ist das sogenannte Distance Vector Routing. Dabei speichern Router in ihren Tabellen nicht nur über welche Schnittstelle andere Rechner erreichbar sind sondern nauch zu welchen Kosten. Die Kosten können dabei als Anzahl der Zwischenstationen, als Verzögerungszeit oder als Durchsatz der Verbindung definiert werden. In regelmäßigen Abständen oder bei Änderungen der Netzwerktopologie, sendet ein Router einen sogenannten Distance Vector an alle seine Nachbarn, der beschreibt, welche Rechner er mit welchen Kosten erreichen kann.

Ein Router, der neu ins Netz kommt, kennt noch keine anderen Rechner und sendet als erstes einen Distance Vector, der nur ihn selbst mit den Kosten 0 enthält, an alle seine Nachbarn. Diese senden daraufhin ihre eigenen Distanzvektoren, mit deren Hilfe der neue Router dann seine Routing-Tabelle erweitern kann. Auf diese Weise halten alle beteiligten Rechner ihre Routing-Tabellen automatisch auf aktuellem Stand. Ausfallende Rechner können erkannt werden, indem jedem Eintrag in der Routing-Tabelle ein Zeitstempel hinzugefügt wird. Sobald eine Route eine bestimmte Zeit nicht mehr aktualisiert wird, wird angenommen, dass sie nicht mehr existiert. In diesem

Fall werden im nächsten Distanzvektor für den entsprechenden Eintrag die Kosten “unendlich” propagiert.

Ein alternatives Verfahren ist sogenanntes link state routing, bei dem jeder Router die gesamte Topologie des Netzwerkes lernt und daraus eigenständig kürzeste Wege berechnen kann. Dazu senden alle Router in regelmäßigen Abständen die Kosten der Verbindungen zu ihren direkten Nachbarn an alle Nachbarn. Diese Information wird von allen Routern weitergeleitet, so dass das Netzwerk mit allen solchen Nachrichten von allen Routern geflutet wird. Dadurch lernen alle Router die Kosten aller direkten Verbindungen und können daraus Gesamtkosten für zusammengesetzte Pfade ausrechnen. Ausfallende Router werden dadurch erkannt, dass ihre direkten Verbindungen zunächst noch von ihren Nachbarn aber nicht mehr von ihnen selbst propagiert werden.

Das Internet Protokoll umfasst unterschiedliche Routing-Verfahren die auf unterschiedlichen Hierarchiestufen angewendet werden und die hier skizzierten Mechanismen verfeinern oder kombinieren.

Zuverlässige Kommunikation über ein unzuverlässiges Netzwerk

Im Internet gibt es zwei gängige Protokolle auf der Transportschicht.

Das User Datagram Protokoll (UDP) ist wie das Internet Protokoll (IP), auf dem es basiert, verbindungslos und unzuverlässig. Daten können verloren gehen, dupliziert werden und in unterschiedlicher Reihenfolge ankommen. UDP stellt jedoch durch Prüfsummen sicher, dass korrumpierte Daten erkannt und verworfen werden. Außerdem ermöglicht es über sogenannte Ports die Kommunikation mit verschiedenen Prozessen über die selbe IP-Adresse. Während auf IP-Ebene Rechner über ihre IP-Adresse angesprochen werden, werden auf UDP-Ebene Prozesse auf Rechnern über eine IP-Adresse und einen Port angesprochen. Dadurch wird es möglich, dass viele verschiedene Anwendungen auf einem Rechner gleichzeitig auf das Internet zugreifen können. UDP wird vor allem von Anwendungen wie Internet-Telefonie verwendet, die auf kurze Verzögerungszeiten Wert legen und gelegentlichen Datenverlust verkraften können.

Das Transmission Control Protocol (TCP) stellt verbindungsorientierte zuverlässige Kommunikation auf Basis des Internet Protokolls bereit. Wie UDP verwendet es Ports, um verschiedene Prozesse auf einem Rechner zu identifizieren und Prüfsummen um korrumpierte Daten zu erkennen. Zwei über TCP verbundene Prozesse können miteinander in beide Richtungen beliebig

große Datenmengen austauschen. Die Daten werden dazu von TCP in IP-Pakete verpackt, die separat verschickt werden.

Dadurch, dass Verbindungen im Internet heterogen (also insbesondere unterschiedlich schnell) sind, müssen Datenpakete von Routern in Puffern zwischengespeichert werden, bevor sie weiter verschickt werden können. Selbst über perfekte Verbindungen könnte also Datenverlust dadurch auftreten, dass der Puffer eines Routers voll ist und ankommende Pakete deshalb verworfen werden.

Um den Verlust von Paketen zu erkennen, erwartet der Absender bei TCP eine Empfangsbestätigung vom Empfänger. Falls diese nach Ablauf einer gewissen Zeit nicht eintrifft, nimmt der Absender an, dass das Paket verloren ging und sendet es erneut. Falls eine Empfangsbestätigung verloren geht, bekommt der Empfänger dadurch Pakete doppelt. Damit er diese als dupliziert erkennen kann, werden Pakete mit einer laufenden Nummer durchnummeriert. Bekommt ein Empfänger zweimal ein Paket mit der selben Nummer, kann er das zweite verwerfen.

Insbesondere bei einer Verbindung mit hohem Durchsatz und hoher Verzögerung ist es ineffizient, immer erst auf eine Bestätigung zu warten, bevor das nächste Paket losgeschickt wird. Bei sogenanntem Pipelining werden mehrere Pakete auf einmal losgeschickt, deren Bestätigungen später nacheinander eintreffen können. Der Empfänger kann die korrekte Reihenfolge der Pakete anhand der laufenden Nummer erkennen. Entweder verwirft er Pakete die außer der Reihe eintreffen, ohne sie zu bestätigen (dann werden sie später vom Absender erneut geschickt) oder er sortiert eintreffende Pakete in einem Empfangspuffer. TCP verwendet die letztere Variante mit einem Empfangspuffer. Außerdem werden Bestätigungen nicht einzeln verschickt sondern kumulativ: eine Bestätigung enthält dazu die nächste erwartete laufende Nummer. Bei Duplex-Kommunikation können Bestätigungen durch sogenanntes Piggybacking mit anderen Datenpaketen kombiniert werden.

Die Internet-Anwendungen DNS, Email und HTTP

IP-Adressen werden als Zahlenkombinationen notiert, die Menschen nur schwer auswendig lernen können. Das Domain Name System (DNS) ermöglicht eine Übersetzung hierarchisch strukturierter Klartextnamen in IP-Adressen auf Basis des User Datagram Protocols (UDP).

Domainnamen bestehen aus einem Hostnamen, möglicherweise mehreren Subdomains, einer Domain und einer Top-Level Domain. Zum Beispiel ist der Name `www.uni-kiel.de` zusammengesetzt aus dem Hostnamen `www`, der Domain `uni-kiel` und der Top-Level Domain `de`.

Das DNS ist ein Internet-Dienst, der es erlaubt solche Namen in zugehörige IP-Adressen zu übersetzen. Dazu muss jede Domain einen Nameserver bereitstellen, der die IP-Adresse von in dieser Domain erreichbaren Hosts kennt oder zumindest andere Nameserver für mögliche Subdomains. Entsprechend gibt es zu jeder Top-Level Domain Nameserver, die Nameserver für die in ihr verwalteten Domains kennen. Um die IP-Adresse zu `www.uni-kiel.de` herauszufinden, könnten wir also zuerst den Nameserver der Top-Level `de` Domain befragen. Dieser würde uns einen Name-Server zur Domain `uni-kiel` nennen, den wir dann nach der IP-Adresse des Hosts `www` fragen könnten.

Dieses Verfahren ist umständlich, da jeder Client mehrere Anfragen stellen und die aktuellen Adressen der Nameserver aller Top-Level Domains kennen müsste. Stattdessen können Clients sogenannte Name Resolver anfragen, die von Internet Service Providern (ISPs) zur Verfügung gestellt werden. Diese kennen die aktuellen Adressen aller Top-Level Domain Name Server und kombinieren mehrere Anfragen um die Anfrage eines Clients zu beantworten. Für häufig angefragte Domainnamen können Resolver auch Antworten zwischenspeichern, um sie nicht immer wieder neu erfragen zu müssen.

Das Domain Name System wird auch verwendet, um Emails vom Mailserver des Absenders zum Mailserver des Empfängers zu transportieren.

Beim Verschicken einer Email kommuniziert der Mailclient des Absenders per Simple Mail Transfer Protocol (SMTP) mit einem Mailserver und dieser dann mit dem Mailserver des Empfängers. Um Mails von seinem Mailserver herunterzuladen, kann der Empfänger das Post Office Protocol (POP) oder das Internet Message Access Protocol (IMAP) verwenden.

Eine Email besteht aus einem Header und einem Rumpf mit dem eigentlichen Inhalt der Mail. Der Header enthält mindestens Felder `From` für den Absender und einen Zeitstempel `Date` und in der Regel auch ein Feld `To` für den Empfänger und eine Feld `Subject` für den Betreff.

Der `Message-Id` Header identifiziert eine Email eindeutig. Dieser Wert kann im `In-Reply-To` Feld verwendet werden um Konversationen kenntlich zu machen. Der `Received` Header ermöglicht es, nachzuvollziehen, welche Mailserver eine Email auf ihrem Weg zum Empfänger weitergeleitet haben.

Ursprünglich wurde das Format für Emails nur für den Austausch von Textdaten im ASCII-Format konzipiert. Heutzutage können auch Emails in anderen Zeichensätzen (zum Beispiel chinesischen) verschickt werden. Auch Bild- und Tondateien können per Email verschickt werden. Dazu werden die Daten so umkodiert, dass existierende Mailserver, die davon ausgehen,

dass die Nachrichten ASCII-Daten enthalten, weiter verwendet werden können.

Als letztes Anwendungsprotokoll im Internet streifen wir das HyperText Transfer Protocol (HTTP). Wie der Name sagt, ist es dazu da, sogenannte Hypertext Dokumente auszutauschen. Hypertext Dokumente enthalten Hyperlink genannte Referenzen auf andere Hypertext Dokumente und bilden so ein Netz von Dokumenten im Internet, das sogenannte World Wide Web (WWW). Hypertext Dokumente werden von Webservern bereitgestellt und von Webbrowsern heruntergeladen.

Hypertext Dokumente sind in der HyperText Markup Language (HTML) verfasst, die wir später genauer kennen lernen werden. Sie werden über sogenannte Universal Resource Identifier (URI) adressiert, die im Fall von Webseiten aus dem Zugriffsschema, einem Domainnamen und einem Pfad bestehen. Der Identifier `http://www.uni-kiel.de/index.html` besteht zum Beispiel aus der Protokollbezeichnung `http://` als Zugriffsschema, dem Domainnamen `www.uni-kiel.de` sowie dem Pfad `/index.html` zum Zugriff auf eine entsprechende Datei auf dem Webserver `www` der Domain `uni-kiel.de`.

URI's können weitere Komponenten enthalten. Zum Beispiel kann dem Pfad nach einem Fragezeichen ein sogenannter Querystring folgen und hinter einer Raute `#` kann der Name eines Fragmentes einer Datei stehen.

Die HyperText Markup Language (HTML)

Eine der wichtigsten Anwendungen des Internet ist das WWW: ein weltweites Netz untereinander verlinkter sogenannter HyperText-Dokumente. Solche Dokumente werden mit Hilfe der Dokumentenbeschreibungs-Sprache HTML (für HyperText Markup Language) definiert, auf Webservern abgelegt und von Webbrowsern mit Hilfe des HyperText Transfer Protokolls (HTTP) von solchen herunter geladen.

HTML-Dokumente sind Textdokumente mit sogenanntem Markup: zusätzlichen Anweisungen zur Strukturierung. Die Struktur eines HTML-Dokuments wird durch sogenannte Tags spezifiziert, die das Dokument hierarchisch in seine Bestandteile zerlegen. Die Struktur eines einfachen HTML-Dokuments ist wie folgt.

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="utf-8">
    <title>Dies ist eine HTML-Seite</title>
```

```
  </head>
  <body>
    ...
  </body>
</html>
```

Die erste Zeile ist eine sogenannte Dokumenttyp-Definition, mit der die verwendete HTML-Version kenntlich gemacht werden kann. Bei Verwendung der aktuellen HMTL-Version genügt die hier gezeigt Angabe `<!DOCTYPE html>`.

Das eigentliche HTML-Dokument ist in die Tags `<html>` und `</html>` eingeschlossen. Tagnamen werden zwischen spitzen Klammern notiert. Das erste Tag heißt öffnendes und das letzte schließendes Tag, wobei schließende Tags stets den gleichen Namen haben wie zugehörige öffnende Tags, diesem aber ein Schrägstrich vorangestellt wird. Durch öffnende Tags begonnene Bereiche müssen in der Regel durch ein schließendes Tag beendet werden.[3] Dabei muss die hierarchische Struktur des Dokumentes abgebildet werden.

[3] Eine Ausnahme dieser Regel ist das gezeigte `<meta>`-Tag zur Angabe des Zeichensatzes.

Ein HMTL-Dokument hat zwei Bestandteile: einen Kopf (abgesetzt durch `<head>` und `</head>`) und einem Rumpf (abgesetzt durch `<body>` und `</body>`). Hier wäre es falsch, das schließende `</head>`-Tag nach dem öffnenden `<body>`-Tag zu notieren, da dies nicht der hierarchischen Struktur des Dokumentes entsprechen würde. Der Kopf enthält Meta-Informationen, wie hier den Zeichensatz und den Titel des Dokumentes, die nicht angezeigt werden.[4] Der Rumpf enthält den eigentlichen Inhalt des Dokumentes, der im Browser angezeigt wird. Im folgenden behandeln wir Tags zur Strukturierung des Rumpfes eines HTML-Dokuments.

[4] Der Titel einer HTML-Seite wird von gängigen Browsern in der Regel in der Titelzeile des Browserfensters angezeigt, aber nicht im angezeigten Dokument selbst.

Überschriften werden durch Tags mit den Namen `h1` bis `h6` (`h` für heading) deklariert. `h1` bezeichnet dabei eine Überschrift erster Ordnung, `h2` eine zweiter Ordnung und so weiter. In Browsern werden Überschriften erster Ordnung in der Regel am größten und fett dargestellt, Überschriften zweiter Ordnung etwas kleiner und so weiter. Die Darstellung von Dokumentbestandteilen ist jedoch explizit *nicht* in HTML spezifiziert. HTML beschreibt nur die Struktur von Dokumenten. *Wie* ein Dokument dargestellt werden soll, kann jedoch gesondert mit CSS (für Cascading Style Sheets) beschrieben werden, wie wir später noch sehen werden.

Absätze werden durch Tags mit dem Namen `p` (für paragraph) begrenzt. In Browsern werden Absätze in der Regel abgesetzt notiert und zwar unabhängig davon, wie sie in der HTML-Datei selbst formatiert beziehungsweise umgebrochen sind. Zur Anzeige eines HTML-Dokumentes im Browser ist nur die durch Tags deklarierte Struktur relevant nicht jedoch die

Formatierung anhand von Zeilenumbrüchen in der HTML-Datei selbst. Bereiche, die genauso wie im Quelltext der HTML-Datei umgebrochen werden sollen, können zwischen Tags mit dem Namen pre (für **pre-formatted text**) deklariert werden. Dies ist zum Beispiel nützlich, um Programmtext in HTML-Dokumenten anzuzeigen.

Die wichtigsten Bestandteile von HTML-Dokumenten im Vergleich zu gewöhnlichen Dokumenten sind **Verknüpfungen** zu anderen Dokumenten. Diese werden durch Tags mit dem Namen a (für anchor) deklariert. Anders als bei den bisher vorgestellten Tags, bestehen Verknüpfungen aus einem angezeigten Teil (dem Verknüpfungstext) und einem nicht angezeigten Teil (dem Verknüpfungßziel). Der Verknüpfungstext wird wie gewohnt zwischen den Tags geschrieben. Das Verknüpfungsziel wird als sogenanntes Attribut des öffnenden Tags mit dem Namen href (für Hyper Reference) notiert. Zum Beispiel Beschreibt Uni Kiel eine Verknüpfung mit dem Text Uni Kiel und dem Ziel http://www.uni-kiel.de. Verknüpfungsziele werden also durch ihren URI spezifiziert. Statt einen vollständigen URI als Verknüpfungsziel anzugeben, kann der Domainname auch weggelassen werden, um ein Dokument der selben Domain zu verlinken. Analog zu Dateisystemen kann auch ein relativer Pfad angegeben werden, der dann ausgehend vom Verzeichnis des aktuellen Dokumentes interpretiert wird.

Tags mit dem Namen a werden nicht nur für Verknüpfungen sondern auch zur Markierung von Bestandteilen des Dokuments verwendet, auf die durch Angabe des Anker-Namens nach # explizit verlinkt werden kann. Mit ... wird ein sogenannter Anker zu dem umfassten Bestandteil des Dokuments deklariert. Wikipedia deklariert Anker für die Bestandteile aller Dokumente. Wir können also zum Beispiel durch <a href="http://de.wikipedia.org/wiki/Hypertext_Markup_Language#HTML-Struktur">HTML-Struktu direkt auf den Abschnitt "HTML-Struktur" des Wikipedia-Eintrags zu HTML verlinken. In Browsern werden Verknüpfungen in der Regel in einer anderen Farbe und unterstrichen dargestellt um sie von normalem Text abzusetzen. Obwohl CSS es erlaubt die Darstellung nach Belieben anzupassen, sollte mit solchen, von vielen internalisierten, Konventionen nicht (oder nur aus sehr guten Gründen) gebrochen werden.

Bilder können in HTML-Dokumente eingebunden werden, indem ihr URI als Attribut src eines Tags mit dem Namen img angegeben wird. Zum Beispiel können wir durch <img src="http://www.uni-kiel.de/home/grafik/kopf-cau-block.gif"> das Logo der Uni-Kiel in ein HTML-Dokument einbinden.[5] Da Bei Bildern kein eigentlicher Inhalt notiert wird, kann das schließende Tag entfallen. Alternativ können wir auch

[5] Bei der Anzeige fremder möglicherweise geschützter Bilder ist auf Grund von Copyright-Bestimmungen Vorsicht geboten. Bilder aus der Wikipedia dürfen unter bestimmten Vorraussetzungen verwendet werden. Bei Fragen konsultieren Sie bitte Ihren Anwalt.

durch einen angehängten Schrägstrich wie in anzeigen, dass das gerade geöffnete Tag direkt wieder geschlossen wird.

Aufzählungslisten werden in HTML durch Tags mit den Namen `ol` (für ordered list) beziehungsweise `ul` (für unordered list) deklariert. Geordnete Listen werden durchnummeriert während in ungeordneten Listen die Einträge durch ein sogenanntes Bullet (zum Beispiel einen kleinen Kreis) kenntlich gemacht werden. Die Einträge selbst werden zwischen Tags mit dem Namen `li` (für list item) notiert. Zum Beispiel wird der folgende HTML-Code zur Deklaration einer drei-elementigen ungeordneten Liste

```
<ul>
  <li>erster Eintrag</li>
  <li>zweiter Eintrag</li>
  <li>dritter Eintrag</li>
</ul>
```

im Browser wie folgt dargestellt:

- erster Eintrag
- zweiter Eintrag
- dritter Eintrag

Tabellen werden durch Tags mit dem Namen `table` deklariert. Sie bestehen aus Zeilen, die zwischen `tr` (für table row) geschrieben werden und wiederum Einträge enthalten, die durch `td` (für table data) kenntlich gemacht werden. Statt `td` kann auch `th` (für table header) verwendet werden, um den Eintrag als Überschrift zu kennzeichnen. Zum Beispiel könnte die folgende Tabelle

```
<table>
  <tr>
    <th>Vorname</th>
    <th>Nachname</th>
  <tr>
  <tr>
    <td>Sebastian</td>
    <td>Fischer</td>
  </tr>
  <tr>
    <td>Frank</td>
    <td>Huch</td>
  </tr>
  <tr>
    <td>Kai</td>
```

```
    <td>Wollweber</td>
  </tr>
</table>
```

im Browser wie folgt dargestellt werden:

Vorname	Nachname
Sebastian	Fischer
Frank	Huch
Kai	Wollweber

Die genaue Formatierung kann durch CSS beeinflusst werden.

Cascading Style Sheets (CSS)

Bei der Deklaration von HTML-Dokumenten wird zwischen deren Struktur (die in HMTL spezifiziert wird) und deren Formatierung unterschieden. Letztere wird in der Formatierungs-Sprache CSS (Cascading Style Sheets) deklariert. Style Sheets werden im Kopf einer HTML-Datei definiert und können auch in gesonderte Dateien ausgelagert werden um sie in mehreren HTML-Dateien wiederzuverwenden. Im folgenden werden beide Varianten dokumentiert.

```
<html>
  <head>
    <link rel="stylesheet" type="text/css" href="format.css">
    <style type="text/css">
      ...
    </style>
  <head>
  <body>
    ...
  </body>
<html>
```

Hier werden zunächst mit Hilfe eines `link`-Tags ein externes Style Sheet `format.css` eingebunden und dann zwischen Tags mit dem Namen `style` weitere Formatierungsangaben gemacht.

Die Formatierungsangaben selbst haben das Format

```
Selektor {
  Eigenschaft1: Wert1;
  Eigenschaft2: Wert2;
  ...
}
```

wobei durch `Selektor` ein oder mehrer Bestandteile eines HTML-Dokumentes ausgewählt werden können, die dann entsprechend der in geschweifte Klammern eingeschlossenen Formatierungsanweisungen dargestellt werden.

Zum Beispiel wird durch die folgende Angabe

```
h1 {
  font-style: italics;
}
```

spezifiziert, dass Überschriften erster Ordnung kursiv dargestellt werden sollen. Es können auch mehrer Selektoren durch Kommata getrennt angegeben werden:

```
h1, h2, h3 {
  font-style: italics;
}
```

Hier werden Überschriften erster bis dritter Ordnung kursiv dargestellt.

Selektoren können auch hierarchisch strukturiert werden. Zum Beispiel bezieht sich der Selektor `p ul` auf alle ungeordneten Listen, die innerhalb von Absätzen stehen. Um zu kennzeichnen, dass nur solche Listen, die *direkt* innerhalb von Absätzen stehen, selektiert werden sollen, können wir `p>ul` als Selektor verwenden. Um den Unterschied der beiden Selektoren zu verdeutlichen betrachten wir den folgenden Absatz einer möglichen HTML-Datei, der geschachtelte ungeordnete Listen enhält.

```
<p>
  <ul>
    <li>
      <ul>
        <li>A.1</li>
        <li>A.2</li>
      </ul>
    </li>
    <li>B</li>
  </ul>
</p>
```

Der Selektor `p ul` selektiert hier beide ungeordneten Listen, während der Selektor `p>ul` nur die äußere selektiert, die direkt unterhalt des Absatzes steht, nicht aber die innere, die innerhalb eines Listeneintrags steht.

Nütliche Angaben zur Formatierung können Sie dem CSS-Kapitel der Seite SELFHMTL entnehmen. Nützlich sind zum Beispiel Angaben zu Rahmen in Tabellen:

```
table {
  border-top: thin solid;
}

table, table th {
  border-bottom: thin solid;
}
```

Quellen und Lesetipps

- Computer Networking: Principles, Protocols and Practice
- SELFHTML: HTML-Dateien selbst erstellen

22

Dynamische Webseiten

Im Kapitel über Netzwerke haben wir das HTML-Format zur Beschreibung von Webseiten kennen gelernt. Bisher waren diese Seiten statisch, das heißt ihr Inhalt war vordefiniert wie in einem Buch. Dynamische Webseiten können unterschiedliche Inhalte anzeigen, die zum Beispiel bei jedem Abruf aktualisierte Daten aus einer Datenbank enthalten. Sie erlauben es auch, ähnlich wie grafische Anwendersoftware, Eingaben von Anwenderinnen zu verarbeiten. Im Folgenden werden wir kennen lernen, auf welche Art HTML-Seiten Benutzereingaben erlauben und wie diese mit Hilfe der Programmiersprache Javascript im Browser verarbeitet werden können. Dabei werden wir HTML-Seiten definieren, deren Inhalt dynamisch von Benutzereingaben abhängt. Javascript dient also der Client-seitigen Webprogrammierung. Server-seitige Webprogrammierung, bei der Webseiten dynamisch vom Webserver erzeugt werden, lernen wir später kennen.

HTML-Formulare für Benutzereingaben

Neben den bisher vorgestellten Dokument-Elementen, können in HTML-Dateien auch Eingabe-Elemente in Formularen definiert werden. Formulareingaben können, wie wir später sehen werden, mit Programmen auf dem Webserver oder auch mit Javascript im Webrowser verarbeitet werden. Zunächst lernen wir jedoch kennen, wie Formulare definiert werden und welche die wichtigsten Eingabe-Elemente sind.

Ein **Formular** wird in HTML mit dem `form` Tag definiert. Innerhalb des `form` Tags dürfen beliebige andere HTML-Elemente stehen, insbesondere auch Elemente für Benutzereingaben.

Das einfachste Element zur Eingabe ist ein **Texteingabefeld**, das wir mit einem `input` Tag definieren können, deren `type` Attribut `text` ist:

```
<form>
    <input type="text" name="message">
</form>
```

Das Attribut name gibt an, unter welchem Namen der eingegebene Text übermittelt werden soll. Das input Tag hat keinen Inhalt und braucht kein schließendes Tag.

Zur Übermittlung der eingegebenen Daten fügen wir dem Formular einen entsprechenden Knopf hinzu. Diesen können wir mit dem input Tag und dem Attribut submit definieren.

```
<form>
    <input type="text" name="message">
    <input type="submit">
</form>
```

Beim Druck auf den **Submit-Knopf** wird die HTML-Seite neu geladen, wobei die Formulareingaben dem URL als sogenannter Query-Parameter angehängt werden. Dieser wird dem URL nach einem Fragezeichen angehängt und könnte zum Beispiel ?message=Hallo lauten, wenn ein Benutzer Hallo in das Eingabefeld eingibt und dann auf den Submit-Knopf drückt.

Bei komplexeren Formularen können strukturierende HTML-Elemente wie Tabellen verwendet werden um die Eingabefelder anzuordnen.

Neben Texteingabefeldern können wir auch **Auswahllisten** definieren, mit denen unter vorgegebenen Texten gewählt werden kann. Auswahllisten werden mit dem Tag select definiert, wobei jede Option durch ein option Tag angegeben wird. Die folgende Auswahlliste erlaubt zum Beispiel die Angabe der Priorität einer Nachricht.

```
<select name="priority">
    <option value="indifferent">egal</option>
    <option value="important">wichtig</option>
    <option value="urgent">dringend</option>
</select>
```

Das Attribut name definiert wieder den Namen unterdem die gewählte Option übermittelt wird. Die value Attribute der option Tags werden als Wert der Eingabe mit dem gegebenen Namen priority übermittelt, wenn die entsprechende Option vor Absenden des Formulars ausgewählt wurde. Eine mögliche Eingabe bei Auswahl der letzten Option wäre also priority=urgent.

Statt mit einer Auswahlliste können wir vorgegebene Eingaben auch mit Hilfe sogenannter **Radio-Buttons** definieren.

Dieser werden als `input` Tags angegeben, deren `type` Attribut den Wert `radio` hat. Radio-Buttons werden von Webbrowsern in der Regel als klickbare Kreise dargestellt.

```
<ol>
    <li>
        <input name="priority" value="indifferent"> egal
    </li>
    <li>
        <input name="priority" value="important"> wichtig
    </li>
    <li>
        <input name="priority" value="urgent"> dringend
    </li>
</ol>
```

Hier verwenden wir eine geornete Liste um die Optionen anzuordnen. Da Radio-Buttons von sich aus keinen anzuzeigenden Text enthalten fügen wir entsprechende Beschriftungen nach jedem Radio-Button ein. Alle Radio-Buttons mit dem selben Namen werden zu einer Gruppe von Optionen zusammen gefasst, von denen nur höchstens eine ausgewählt werden kann. Die übermittelten Daten entsprechen hier also denen, die auch bei der vorherigen Auswahlliste übermittelt würden.

Schließlich können wir auch noch **Checkboxen** definieren, die als kleine Kästchen dargestellt werden, in denen Häkchen gesetzt werden können.

```
<input type="checkbox" name="confirmation">
```

Dieses Element definiert eine Checkbox, die, wenn ein entsprechendes Häkchen gesetzt wird, beim Absenden des Formulars als `confirmation=on` übermittelt wird.

Im folgenden Kapitel werden wir sehen, wie wir Formulareingaben aus den hier definierten Elementen mit Javascript im Browser verarbeiten können.

Client-seitige Webprogrammierung mit Javascript

Javascript ist eine Programmiersprache, die in einem Webbrowser ausgeführt werden kann. Wir können Javascript Programme ähnlich wie Stylesheets in eine HTML-Datei einbinden: entweder direkt innerhalb von `script` Tags im Header einer HTML Datei oder durch Angabe des URL einer Javascript Datei.

Javascript Code kann in HTML-Dateien in `<script>`-Tags eingebunden werden und dabei entweder in die HTML-Datei selbst geschrieben werden oder aus einer Datei mit der Endung `.js` geladen werden.

```
<script type="text/javascript" src="dateiname.js"></script>
```

Die Sprachelemente von Javascript sind die gewöhnlicher imperativer Programmiersprachen. Wir werden einige im Folgenden exemplarisch immer dann einführen, wenn wir sie benötigen.

Die folgende HTML-Datei verwendet Javascript, um dynamisch die URL unter dem sie erreichbar ist anzuzeigen.

```
<!DOCTYPE html>
<html>
    <head>
        <meta charset="utf-8">
        <title>URL Anzeige</title>
    </head>
    <body>
        <script type="text/javascript">
            document.write(window.location);
        </script>
    </body>
</html>
```

Die Methode `document.write` wird hier verwendet, um die in `window.location` gespeicherte Zeichenkette in das Dokument einzubauen.

Mit `window.location.search` kann auf den sogenannten *query parameter*, also den Teil der URL ab dem Fragezeigen, zugegriffen werden. In Kombination mit der Methode `substring`, die einen Teilstring ab einer gegebenen Position selektiert, können wir den Teil der URL *hinter* dem Fragezeichen mit `window.location.search.substring(1)` abfragen.

Die folgende HTML-Datei wandelt diesen Teil der URL in eine Zahl um und fügt dann in einer Schleife Zahlen von der gegeben Zahl bis eins in das Dokument ein.

```
<!DOCTYPE html>
<html>
<head>
    <meta charset="utf-8">
    <title>Countdown</title>
</head>
<body>
<script type="text/javascript">
var counter = parseInt(window.location.search.substring(1));

while (counter > 0) {
    document.write("<p>" + counter + "</p>");
```

```
    counter = counter - 1;
}
</script>
</body>
</html>
```

Hierzu verwenden wir eine `while`-Schleife, die in Javascript eine etwas andere Syntax hat als in Ruby.

Das Document Object Model (DOM)

Eine Besonderheit von Javascript sind bereitgestellte Objekte zum Zugriff auf die Elemente eines HTML-Dokuments. Am einfachsten kann auf HTML-Elemente zugegriffen werden, wenn diese mit einem `id` Attribut versehen werden. Zum Beispiel kann auf eine wie folgt definierte ungeordnete Liste

```
<ul id="list"></ul>
```

in Javascript durch den Methodenaufruf

```
var list = document.getElementById("list");
```

zugegriffen werden. Das Ergebnis dieses Aufruf, ist ein Javascript Objekt, das verwendet werden kann, um auf Eigenschaften der Liste zuzugreifen oder dazu um diese zu manipulieren. Zum Beispiel können wir der Liste neue Items hinzufügen, wie wir später sehen werden.

Um mit Javascript auf die Elemente eines Dokumentes zugreifen zu können, darf das Programm erst aufgerufen werden, wenn die Seite komplett geladen ist. Dies können wir durch einen Eventhandler erreichen, den wir dem `body` Tag wie folgt zuordnen können.

```
<body onload="processForm();">
    ...
</body>
```

Falls eine Javascript-Funktion `processForm` definiert ist, wird diese aufgerufen, nachdem das Dokument geladen wurde.

Eine solche Funktion können wir wie folgt definieren.

```
function processForm() {
    var query = window.location.search;
    if (query != "") {
        ...
    }
}
```

Hierdurch wird mit dem Schlüsselwort `function` eine Funktion mit dem Namen `processForm` ohne Argumente definiert. Im Rumpf dieser Funktion wird der Query-Parameter des URL mit Hilfe der Eigenschaft `window.location.search` abgefragt und in der Variablen `query` gespeichert. Eine bedingte anweisung testet, ob der Query-String leer ist. Falls nicht, wird der Rumpf der bedingten Anweisung ausgeführt, in dem wir die Formulareingaben verarbeiten können.

Zur Verarbeitung der Formulareingaben benötigen wir einige Funktionen auf Zeichenketten, die wir im Folgenden exemplarisch einführen. Mit der folgenden Anweisung erzeugen wir aus dem Query-String ein Array, dass Formulareingaben enthält.

```
var params = query.substring(1).split("&");
```

Wenn zum Beispiel in der Variablen `query` die Zeichenkette `"?message=Hallo&priority=urgent"` gespeichert ist, hat das Array `params` nach dieser Anweisung den Wert `["message=Hallo", "priority=urgent"]`.

Wir verwenden hier die Methode `substring`, die einen Teilstring ab der gegebenen Position selektiert. Da das erste Zeichen `?` an Position Null steht, wird es durch diesen Aufruf abgeschnitten. Die Methode `split` zerlegt einen String anhand des gegebenen Trennzeichens, hier `"&"`. Als Ergebnis wird ein Array der Teilstrings zurück gegeben, die zwischen den Trennzeichen stehen.

Wir können nun das Array `params` in einer Schleife durchlaufen und in jedem Schritt dem in der Variablen `list` gespeicherten HTML-Element ein Item hinzufügen.

```
for (var i = 0; i < params.length; i++) {
    var item = document.createElement("li");
    var text = document.createTextNode(params[i]);

    item.appendChild(text);
    list.appendChild(item);
}
```

Hier verwenden wir eine `for`-Schleife, die sich deutlich von einer Zahlschleife in Ruby unterscheidet. Der Kopf der `for`-Schleife definiert eine Variable `i`, die alle Indizes des Arrays `params` durchläuft. Die beiden ersten Zeilen des Schleifenrumpfes verwenden Methoden zum Erzeugen von HTML-Elementen und Text-Knoten. In der Variablen `item` wird ein neu erzeugtes `li` Element gespeichert, die Variable `text` speichert einen neuen Text-Knoten mit dem aktuellen Name-Wert-Paar der Formulareingabe.

Die beiden letzten Zeilen des Schleifenrumpfes verwenden die Methode `appendChild`, um das neu erzeugte List-Item der in der Variablen `list` gespeicherten Liste hinzuzufügen.

Wenn wir nach Definition der Funktion `processForm` das definierte Formular ausfüllen und absenden werden daraufhin der in der selben Seite enthaltenen Liste die Formulareingaben hinzugefügt.

23

Web-Programmierung mit Ruby

In einem früheren Kapitel haben wir HTML zur Beschreibung von Webseiten kennengelernt und später auch dynamische Webseiten mit Hilfe von Javascript definiert, die im Browser ausgeführt werden. Neben dieser Client-seitigen Web-Programmierung gibt es auch Server-seitige Web-Programmierung. Dabei werden die dynamischen Anteile nicht im Browser ausgeführt sondern auf dem Webserver, so dass Webseiten aus dort (zum Beispiel in einer Datenbank) gespeicherten Daten generiert werden können.

HTML mit Ruby generieren

Wir wollen im Folgenden einen Web-Server in Ruby implementieren. Zunächst sehen wir uns dazu an, wie wir HTML-Quelltext in Ruby generieren können. Da HTML-Quelltext Text ist, können wir ihn in Ruby als Zeichenkette darstellen. Zum Beispiel liefert die folgende Funktion ein `<h1>`-Tag mit übergebenem Inhalt dargestellt als Zeichenkette zurück.

```
def heading title
  return "<h1>#{title}</h1>"
end
```

Wir können diese Funktion in `irb` mit dem folgenden Aufruf testen.

```
irb> heading 'Hallo'
 => "<h1>Hallo</h1>"
```

Wenn wir als Argument der `heading` Funktion HTML-Quelltext übergeben wird dieser unverändert in das Ergebnis eingebaut.

```
irb> heading '</h1><script>fire_missiles();</script><h1>'
 => "<h1></h1><script>fire_missiles();</script><h1></h1>"
```

Dies kann in Kombination mit Benutzereingaben zu Sicherheitsproblemen führen. Später sehen wir, was wir dagegen tun können.

Zunächst wollen wir ein etwas komplexeres HTML-Fragment definieren. Die folgende Funktion erzeugt aus einer übergebenen Liste von Zeichenketten eine ungeordnete Liste mit entsprechenden Einträgen.

```
def todo_each items
  result = "<ul>"
  items.each do |item|
    result = result + "<li>#{item}</li>"
  end
  result = result + "</ul>"
  return result
end
```

Das Ergebnis wird hier schrittweise in der Variablen `result` zusammengbaut und am Ende des Funktionsrumpfes zurückgegeben. Der folgende Aufruf dokumentiert die Verwendung der definierten Funktion.

```
irb> todo ['essen', 'lesen']
 => "<ul><li>essen</li><li>lesen</li></ul>"
```

Unter Verwendung der `Array`-Methode `collect`, können wir das Ergebnis auch direkt als Ruby-Ausdruck definieren, ohne schrittweise eine Hilfsvariable zu manipulieren.

```
def todo_collect items
  return "<ul>#{items.collect do |item|
    "<li>#{item}</li>"
  end.join}</ul>"
end
```

Die Methode `join` fasst hier das von `collect` berechnete Array zu einer einzigen Zeichenkette zusammen, die dann zwischen `<ul>`-Tags eingefügt wird.

Gegenüber einer Definition in HTML wirken die gezeigten Funktionen vergleichsweise kompliziert. Zum einen kommt das daher, dass sie durch die Parametrisierung allgemeiner sind als der konkrete HTML-Quelltext, der durch einen Aufruf der definierten Funktion entsteht. Aber auch das explizite Hantieren mit Zeichenketten verkompliziert die Definitionen.

Mit Hilfe des Gems `html` können wir die gezeigten Funktionen noch leserlicher definieren. Die Funktion zum Erzeugen der Überschrift sieht damit zum Beispiel wie folgt aus.

```
def heading_html title
  HTML.fragment {
    h1 { text title }
  }
end
```

Die Funktion `HTML.fragment` liefert eine Zeichenkette zurück, die die im übergebenen Block definierten Elementen enthält. Für jedes denkbare HTML-Element ist dabei eine entsprechende Prozedur definiert, die wir verwenden können, um Elemente im Rückgabewert zu erzeugen. Hier ist ein Beispielaufruf der definierten Funktion.

```
irb> heading_html 'Hallo'
 => "<h1>Hallo</h1>"
```

Durch Verwendung des `html`-Paketes wird nun HTML-Quelltext im Argument der Funktion anders behandelt als bei unserer vorherigen Definition.

```
irb> heading_html '</h1><script>fire_missiles();<script><h1>'
 => "<h1>&lt;/h1&gt;&lt;script&gt;fire_missiles();&lt;script&gt;&lt;h1&gt;</h1>"
```

Alle Sonderzeichen werden HTML-spezifisch so umgewandelt, dass der Titel später im Browser genau so aussieht, wie die Zeichenkette, die wir übergeben haben.

Die Funktion zum Erzeugen einer Todo-Liste können wir wie folgt anpassen.

```
def todo_html items
  HTML.fragment {
    ul {
      items.each do |item|
        li { text item }
      end
    }
  }
end
```

Hier werden mit Hilfe der `each`-Methode auf Arrays im Block, der an `ul` übergeben wird, mehrere Aufrufe von `li` ausgeführt. Wir können also beliebige Ruby-Konstrukte verwenden, um Anweisungen zur HTML-Generierung zu definieren. Die Implementierung mit Hilfe des `html`-Paketes stellt sicher, dass alle Elemente korrekt geschachtelt sind und dass schließende Tags zu den entsprechenden öffnenden Tags passen.

Auch Elemente mit Attributen lassen sich auf diese Weise erzeugen. Dazu passen wir die Zeile im Block, der im obigen Beispiel an `each` übergeben wird, wie folgt an.

```
  li { a(href: '#' + item) { text item } }
```

Attribute werden also als Hash-Parameter übergeben. Der Aufruf `todo_html ['essen','lesen']` erzeugt jetzt die folgende Zeichenkette (ohne Umbrüche).

```
<ul>
  <li><a href='#essen'>essen</a></li>
  <li><a href='#lesen'>lesen</a></li>
</ul>
```

Schließlich wollen wir nun die beiden definierten Funktionen verwenden um ein komplettes HTML-Dokument zu generieren.

```
def page
  HTML.doc(charset: 'utf-8') {
    head { title { text 'Todo' } }
    body {
      inline heading_html 'Todo'
      inline todo_html ['essen', 'lesen']
    }
  }
end
```

Das `charset`-Attribut wird dem von `HTML.doc` erzeugten öffnenden `<html>`-Tag hinzugefügt. Die Prozedur `inline` des `html`-Paketes fügt (anders als `text`) die übergebene Zeichenkette unverändert in das erzeugte Dokument ein, dass (mit ein paar Umbrüchen angereichert) wie folgt aussieht.

```
<!doctype html>
<html charset="utf-8">
  <head><title>Todo</title></head>
  <body>
    <h1>Todo</h1>
    <ul>
      <li><a href="#essen">essen</a></li>
      <li><a href="#lesen">lesen</a></li>
    </ul>
  </body>
</html>
```

HTTP Anfragen mit Ruby beantworten

Um ein Ruby-Programm zu schreiben, dass HTTP-Anfragen mit generiertem HTML-Quelltext beantwortet, verwenden wir das Web-Framework Sinatra. Nachdem wir es mit `require`

'sinatra' in unser Programm eingebunden haben, sorgt die folgende Anweisung dafür, dass Anfragen an den gestarteten Server mit der oben definierten Seite beantwortet werden, wenn der Pfad der zugehörigen URL / ist (oder es keinen Pfad gibt, also auf das Wurzelverzeichnis bzw. die Homepage zugegriffen wird).

```
get '/' do
  page
end
```

Der get-Aufruf sorgt dafür, dass das gestartete Programm HTTP-GET Anfragen an den Pfad / beantwortet. Bei jeder solchen Anfrage wird der übergebene Block ausgeführt, der als Ergebnis die von page generierte HTML-Seite hat. Dieses Ergebnis des Blocks wird als Antwort an den Client geschickt, der die HTTP-GET-Anfrage gestellt hat.

Wir können das Programm wie folgt mit ruby im Terminal starten, wenn wir es in einer Datei server.rb abspeichern.

```
# ruby server.rb
```

Dieser Aufruf startet einen HTTP-Server auf Port 4567. Wir können also in einem Webbrowser über die URL localhost:4567 eine Anfrage stellen und bekommen dann die Todo-Liste angezeigt.

Web-Anwendung zur Verwaltung der Filmdatenbank

Wir wollen nun eine Web-Anwendung schreiben, die es erlaubt, auf unsere früher entwickelte Filmdatenbank zuzugreifen. Zur Erinnerung: Diese Datenbank enthält drei Tabellen.

- Die Tabelle movie hat die Attribute id, title, year und director, wobei das letzte Attribut Fremdschlüssel in die Tabelle person enthält.
- Die Tabelle person hat die Attribute id und name.
- Die Tabelle actor hat die Attribute id, person und movie, wobei die beiden letzten Attribute Fremdschlüssel in die entsprechenden Tabellen enthalten.

Zum Zugriff auf die Datenbank verwenden wir die früher entwickelte Datei database_sqlite.rb mit Hilfsobjekten zum Zugriff auf SQLite-Datenbanken. Unser Programm movie_app.rb beginnt deshalb wie folgt.

```
require 'html'
require 'sinatra'
require_relative 'database_sqlite'

def db
  return Database.new 'movies.sqlite'
end
```

Die Funktion `db` verwenden wir später, um bei der Beantwortung von Anfragen an den Server ein Objekt zum Zugriff auf die Filmdatenbank zu erzeugen.

Anfragen an den Wurzelpfad beantwortet unsere Anwendung mit einem Link zur Liste aller Filme.

```
get '/' do
  page 'Filmdatenbank', HTML.fragment {
    a(href: '/movies') { text 'Liste aller Filme' }
  }
end
```

Die Funktion `page` erzeugt eine HTML-Datei mit übergebenem Titel und Inhalt.

```
def page page_title, contents
  HTML.doc(charset: 'utf-8') {
    head { title { text page_title } }
    body {
      h1 { text page_title }
      inline contents
    }
  }
end
```

Damit HTTP-GET-Anfragen an den Pfad `/movies` von unserer Anwendung beantwortet werden, fügen wir ihr den folgenden Aufruf hinzu.

```
get '/movies' do
  page 'Alle Filme', movie_list(db['movie'].rows)
end
```

Auch dieser Block verwendet die oben definierte Funktion `page`. Der Inhalt der Seite wird mit der folgenden Funktion erzeugt.

```
def movie_list movies
  HTML.fragment {
    ul {
```

```
      movies.each do |movie|
        li {
          a(href: "/movies/#{movie['id']}") {
            text movie_display movie
          }
          inline delete_button 'entfernen', "/movies/#{movie['id']}"
        }
      end
      li {
        form(method: 'post') {
          input(type: 'text', name: 'title', placeholder: 'Titel')
          input(type: 'text', name: 'director', placeholder: 'Regisseur')
          input(type: 'number', name: 'year')
          input(type: 'submit', value: 'Speichern')
        }
      }
    }
  }
end
```

Zu jedem Film wird mit Hilfe der Funktion `movie_display` der Titel und das Erscheinungsjahr angezeigt.

```
def movie_display movie
  return "#{movie['title']} (#{movie['year']})"
end
```

Jeder Eintrag ist verlinkt zu einer Detail-Ansicht für Filme, deren Pfad die `id` des entsprechenden Films enthält. Zusätzlich wird hinter jedem Eintrag ein Knopf zum Löschen eingefügt, der durch die Funktion `delete_button` erzeugt wird, die wir später besprechen. Am Ende der Film-Liste definieren wir ein Formular zur Eingabe der Stammdaten eines neuen Filmes.

Der Link zur Detailansicht, der Knopf zum Löschen und das Formular zum Anlegen neuer Filme lösen alle neue HTTP-Anfragen aus, die wir mit unserem Programm beantworten müssen. Zunächst sehen wir uns die GET-Anfrage zur Detailansicht von Filmen an.

```
get '/movies/:movie_id' do |movie_id|
  movie = db['movie'][movie_id]
  movie['director'] = db['person'][movie['director']]
  movie['actors'] = movie_actors movie

  page movie_display(movie), movie_details(movie)
end
```

Hier steht im Pfad ein sogenannter Platzhalter `:movie_id`. Dieses sogenannte Pfad-Muster passt also auf viele verschiedene Pfade. Die übergebene `id` wird dem behandelnden Block als

Variable movie_id übergeben. Dieser Block erzeugt zunächst ein Hash-Objekt movie, das Daten enthält, die später in der Detailansicht angezeigt werden sollen. Der Datensatz aus der Tabelle movie wird dazu durch weitere Anfragen um person-Datensätze für Regisseur und Schauspieler erweitert.

Die Funktion movie_actors liefert ein Array von person-Datensätzen der Schauspieler eines Films.

```
def movie_actors movie
  return db['actor'].all_where('movie = ?', movie['id']).collect do |entry|
    db['person'][entry['person']]
  end
end
```

Die Funktion movie_details erzeugt aus den abgefragten Daten eine HTML-Seite, die deise anzeigt.

```
def movie_details movie
  HTML.fragment {
    p { text "von #{movie['director']['name']}" }
    p {
      text 'mit'
      ul {
        movie['actors'].each do |actor|
          li { text actor['name'] }
        end
      }
    }
    p {
      a(href: '/movies') { text 'alle Filme' }
    }
  }
end
```

Neben dem Regisseur und einer Liste von Schauspielern zeigt diese Seite auch einen Link an, der auf die Liste aller Filme zurück verweist.

Wir haben noch nicht geklärt, wie POST-Anfragen behandelt werden, die vom Formular zum Hinzufügen von Filmen gesendet werden. Der folgende Aufruf ist dafür zuständig.

```
post '/movies' do
  movie = {
    'title' => params['title'],
    'year' => params['year'].to_i,
    'director' => db['person'].insert('name' => params['director'])
  }
  db['movie'].insert movie
```

```
  redirect to '/movies'
end
```

Der übergebene Block erzeugt zunächst einen neuen Datensatz aus den übertragenen Formulareingaben. Diese sind in Sinatra Anwendungen über das Objekt `params` verfügbar. Da im Formular der Name von Regisseuren steht, sorgen wir mit Hilfe der `insert`-Methode dafür, dass ein entsprechender Datensatz in der `person`-Tabelle vorhanden ist. Die `insert`-Methode fügt keinen neuen Datensatz ein, falls schon ein identischer Datensatz existiert und liefert in jedem Fall die `id` des eingefügten Datensatzes zurück. Wir verwenden sie als Attributwert des Fremdschlüssels `director` im Datensatz für die `movie`-Tabelle. Nachdem der so erzeugte Datensatz in die `movie`-Tabelle eingefügt wurde, senden wir als Antwort an den HTTP-Client eine sogenannte Weiterleitungs-Antwort. Diese fordert den Client auf, eine neue GET-Anfrage an die mitgegebene URL zu stellen. Dadurch wird hier die Seite mit der Liste aller Filme neu geladen, die anschließend den neu hinzugefügten Film anzeigt.

Schließlich diskutieren wir noch wie der Knopf zum Löschen von Filmen erzeugt und die zugehörige Anfrage behandelt wird. Die Funktion `delete_button` erzeugt einen Knopf, mit dem DELETE-Anfragen gesendet werden können.

```
def delete_button label, url
  HTML.fragment {
    form(method: 'post', action: url, style: 'display: inline') {
      input(type: 'hidden', name: '_method', value: 'delete')
      input(type: 'submit', value: label)
    }
  }
end
```

Da Webbrowser ohne Javascript nur GET- und POST-Anfragen senden können, sendet der Knopf tatsächlich eine POST-Anfrage statt einer DELETE-Anfrage. Durch das versteckte Eingabefeld mit dem Namen `_method` dessen Wert `delete` ist, behandelt der Webserver diese Anfrage jedoch genau wie eine DELETE-Anfrage. Wir reagieren darauf wie folgt.

```
delete '/movies/:movie_id' do |movie_id|
  db.execute('delete from actor where movie = ?;', movie_id)
  db['movie'][movie_id] = nil

  redirect to '/movies'
end
```

Im übergebenen Block werden zunächst aus der `actor`-Tabelle alle Datensätze gelöscht, die auf den zu löschenden `movie`-Datensatz verweisen. Anschließend wird der über `movie_id` referenzierte Datensatz aus der `movie`-Tabelle gelöscht.

Eine weitere gängige HTTP-Methode, die von Webbrowsern jedoch nicht unterstützt wird, ist PUT. Diese Methode wird verwendet, um Daten hinter einer URL zu ersetzen. Wir können sie mit Hilfe von Formularen analog zur obigen DELETE-Anfrage mit einem versteckten Eingabefeld mit Namen `_method` und Wert `put` erzeugen. In der Server-Anwendung behandeln wir solche Anfragen dann entsprechend durch einen Aufruf der Prozedur `put`. Zum Beispiel könnten wir eine PUT-Anfrage zum Bearbeiten der Stammdaten eines Filmes wie folgt behandeln.

```
put '/movies/:movie_id' do
  # ...
end
```

Quellen und Lesetipps

- Website zu Sinatra

24
Informationstheorie und Daten-Kompression

Informationstheorie beschäftigt sich mit dem Informationsgehalt von Nachrichten. Ein wichtiges Ergebnis ist eine theoretische Grenze für den Kompressionsgrad bei verlustfreier Kompression von Daten. Der zentrale Begriff zum Verständnis dieser Grenze ist der der **Entropie** für den **mittleren Informationsgehalt** oder die **Informationsdichte** einer Nachricht. Wir werden im Folgenden diese Begriffe beispielhaft klären und anschließend ein Kompressionsverfahren kennen lernen, dass den theoretisch bestmöglichen Kompressionsgrad erreicht.

Entropie als Informationsdichte

Der Begriff Nachricht kann dabei weit gefasst werden. Zum Beispiel kann die Verkündung eines Wahlergebnisses als Nachricht aufgefasst werden. In der Regel werden Wahlen durchgeführt, wenn nicht vorher bekannt ist, wie sie ausgehen. In diesem Fall hat das Wahlergebnis relativ hohen Informationssgehalt. Wird hingegen kurz nach einer Wahl eine weitere durchgeführt, kann davon ausgegangen werden, dass sich das Ergebnis nicht sehr vom vorherigen unterscheidet. Der Informationsgehalt in diesem Fall wäre relativ gering.

Als weiteres Beispiel einer Nachricht können wir die Ergebnisse fortgesetzter Münzwürfe betrachten. Wenn die Ergebnisse *Kopf* und *Zahl* mit gleicher Warscheinlichkeit auftreten, hat die Nachricht der Wurfergebnisse eine Informationsdichte von einem Bit pro Münzwurf, da zur Darstellung jedes Ergebnisses ein Bit benötigt wird. Hat die Münze hingegen Zahlen auf beiden Seiten (ist die Warscheinlichkeit für *Zahl* also gleich eins), so enthält das Ergebnis der Münzwürfe keine Information und auch seine Informationsdichte ist null.

Zur Darstellung natürlichsprachlichen Texts haben wir bereits den ASCII-Code kennen gelernt, bei dem jedes Zeichen mit acht Bit kodiert wird. Die Informationsdichte ist hier jedoch geringer als acht Bit pro Zeichen, da in natürlichsprsachlichen

Texten einige Zeichen häufiger vorkommen als andere. Die unterschiedlichen Häufigkeiten einzelner Buchstaben erlauben es, natürlichsprachlichen Text deutlich kompakter zu kodieren als im ASCII-Code, ohne dabei Information zu verlieren.

Bei verlustfreier Kompression von Daten bleibt der Informationsgehalt einer Nachricht gleich, während ihre Länge abnimmt. Die Informationsdichte nimmt dabei also zu. Dadurch ergibt sich, wie wir später sehen werden, eine theoretische Grenze für den maximal erreichbaren Kompressionsgrad. Zunächst sei nur darauf hingewiesen, dass das Ergebnis von Wurfergebnissen einer fairen Münze (die also *Kopf* und *Zahl* mit gleicher Warscheinlichkeit liefert) verlustfrei nicht mit weniger als einem Bit pro Münzwurf dargestellt werden kann, während die Wurfergebnisse einer Münze, die immer *Zahl* liefert, ohne Bits dargestellt werden können.

Optimale Kompression natürlichsprachlichen Texts

Der Schlüssel zur optimalen Kompression natürlichsprachlichen Texts ist es, unterschiedliche Zeichen mit Binärcodes unterschiedlicher Länge zu kodieren. Häufig vorkommende Zeichen bekommen eine kurze Kodierung während für seltene Zeichen eine längere verwendet werden kann. Dadurch sinkt die durchschnittliche Anzahl Bits, die pro Zeichen benötigt werden, die Informationsdichte steigt also.

Da die Länge der Binärkodierung eines Zeichens vom Zeichen selbst abhängt, können so kodierte Texte nicht so einfach dekodiert werden wie im ASCII-Code kodierte Texte. Um Binärcodes unabhängig von ihrer Lange eindeutig erkennen zu können, dürfen sie nicht willkürlich gewählt werden. Kein Binärcode darf Präfix eines anderen sein, da dann nicht klar ist, wann ein Codewort endet. Ist der Code hingegen **präfixfrei**, kein Codewort also Präfix eines anderen, so kann eindeutig bestimmt werden, wann die Kodierung eines Zeichens zuende ist.

Codes fester Länge sind immer präfixfrei. Im Allgemeinen können wir präfixfreie Codes als Binärbaum darstellen, deren Blätter mit den kodierten Zeichen beschriftet sind, wie das folgende Beispiel zeigt. Wollen wir den Text `eine leise eselei` in einem Code fester Lange kodieren, so benötigen wir pro Zeichen drei Bit, da er (inklusive Leerzeichen) sechs verschiedene Zeichen enthält. Im folgenden Binärbaum ist ein möglicher Code fester Lange dargestellt, mit dem dieser Text kodiert werden könnte.

Das Zeichen `e` wird danach durch die Bitfolge `000` kodiert, das Zeichen `l` durch `001`, `s` durch `010` und so weiter entsprechend der Kantenbeschriftungen im Baum. Die Codewörter `110`

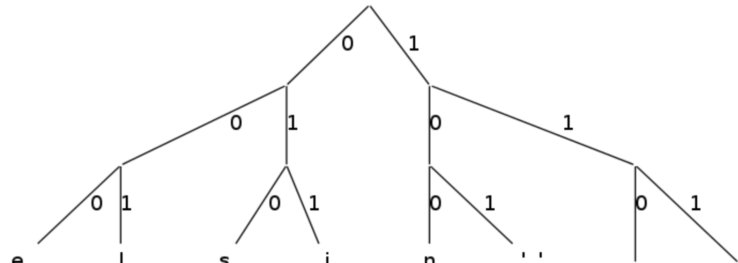


Abbildung 24.1: Möglicher Code fester Länge zur Kodierung des Textes `eine leise eselei`

und `111` werden hier nicht benötigt, da nur sechs verschiedene Zeichen kodiert werden. Da der zu kodierende Text insgesamt aus 17 Zeichen besteht, hat die Kodierung mit drei Bit pro Zeichen eine Länge von $3 \cdot 17 = 51$ Bit. Durch geschickte Wahl eines präfixfreien Codes variabler Länge, können wir den selben Text mit nur 40 Bit kodieren, im Mittel also mit etwa $2{,}35$ Bit pro Zeichen. Dazu ordnen wir, entsprechend dem folgenden Binärbaum, häufigen Zeichen eine kürzere Kodierung zu als seltenen.

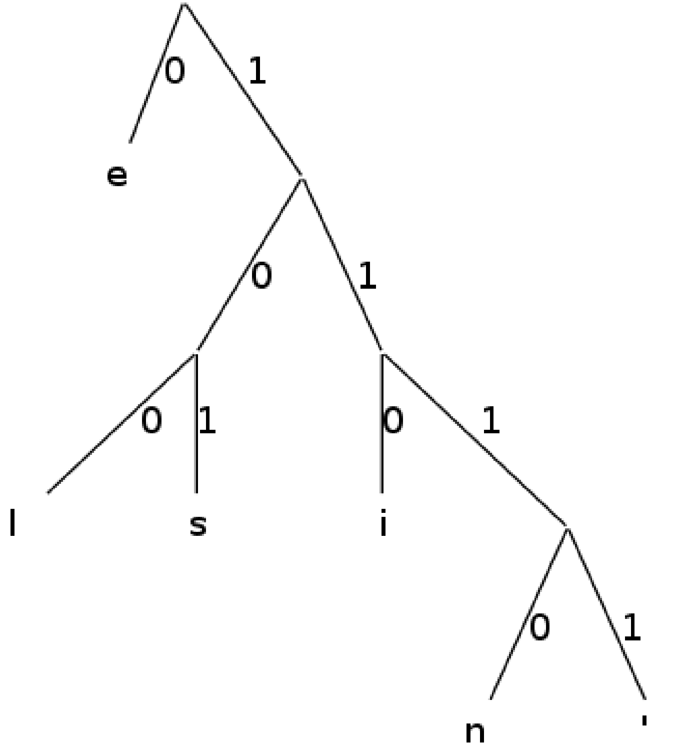


Abbildung 24.2: Optimaler präfixfreier Code variabler Länge zur Kodierung des Textes `eine leise eselei`

Gemäß dieses Codes wird das Zeichen `e` durch ein einzelnes Bit `0` kodiert, `l` durch `100`, `s` durch `101` und so weiter. Das Zeichen `n` und das Leerzeichen haben sogar eine Kodierung der Länge vier. Trotzdem ist die Gesamtlänge des kodierten Textes kürzer als mit einem Code fester Länge, da

das Zeichen `e` mit kurzer Kodierung deutlich häufiger vorkommt als das Zeichen `n` und das Leerzeichen. Insgesamt ergibt sich als Kodierung von `eine leise eselei` die Bitfolge `01101110011111000110101011110101010000110`.

Da die verwendete Komprimierung optimal ist, entspricht die Größe 2,35 Bit pro Zeichen der Entropie oder Informationsdichte des Textes `eine leise eselei`, vorausgesetzt die Häufigkeiten der in der Nachricht vorkommenden Zeichen sind bekannt. In der Praxis muss man entweder diese Häufigkeiten (bzw. den aus ihnen berechneten Code) zusätzlich übermitteln oder zur Konstruktion des Codes relative Buchstabenhäufigkeiten der verwendeten Sprache verwenden, so dass der Code nicht von der Nachricht abhängt. Im letzteren Fall ist die Kodierung dann möglicherweise nicht mehr optimal, wenn die Buchstabenhäufigkeiten im Text von der in der verwendeten Sprache abweichen.

Berechnung optimaler Codes als Huffman-Baum

Zu einem gegebenen Text kann ein optimaler präfixfreier Code variabler Länge automatisch berechnet werden. Wir werden den dazu verwendeten Algorithmus anhand des obigen Beispiels erläutern, dabei also den gezeigten Code in seiner Binärbaumdarstellung erzeugen. Dieser Baum wird nach dem Erfinder des Algorithmus zu seiner Berechnung auch Huffman-Baum genannt.

Um einen Huffman-Baum zu berechnen, bestimmen wir zunächst die Häufigkeiten der verschiedenen Zeichen der zu kodierenden Nachricht. Für die Nachricht `eine leise eselei` ergeben sich die folgenden Häufigkeiten.

Zeichen	Häufigkeit	Zeichen	Häufigkeit
`e`	7	`s`	2
`i`	3	`' '`	2
`l`	2	`n`	1

Diese Paare aus Zeichen und Häufigkeiten bilden die Blätter des erzeugten Huffman-Baums. Die inneren Knoten entstehen durch schrittweise Kombination von kleineren Bäumen zu größeren. Dazu verfahren wir nach dem folgenden Algorithmus.

```
Erzeuge Huffman-Baum aus Text:
    Berechne Häufigkeiten der Zeichen in Text
    Erzeuge daraus sortierte Liste von Blättern

    Solange diese Liste mehr als ein Element enthält
```

```
        Entferne zwei Knoten mit geringsten Häufigkeiten
        Erzeuge Teilbaum aus diesen Knoten
        Beschrifte ihn mit Summe der Häufigkeiten
        Füge ihn in die sortierte Liste ein

    Gib einziges Element der Liste zurück
```

Für unser Beispiel verfahren wir also wie folgt.

- Wir kombinieren zunächst das Zeichen `n` mit dem Leerzeichen zu einem neuen Knoten mit Gesamthäufigkeit drei.[1]
- Dann kombinieren wir die nun mit den geringsten Häufigkeiten beschrifteten Knoten für `l` und `s` zu einem Knoten mit Gesamthäufigkeit vier.
- Nun kombinieren wir den zuerst erzeugten Knoten mit dem für das Zeichen `i` mit Gesamthäufigkeit sechs.
- Diesen kombinieren wir mit dem erzeugten Knoten der Häufigkeit vier zu einem Knoten der Gesamthäufigkeit zehn.
- Die beiden verbleibenden Knoten kombinieren wir nun zum Endergebnis aus dem zuletzt erzeugten Knoten und dem Zeichen `e`.

[1] Wir könnten das Zeichen `n` auch mit einem anderen Zeichen der Häufigkeit zwei kombinieren, wodurch ein anderer, ebenfalls optimaler, Code entstehen würde. Auch welchen Knoten wir links und welchen rechts anhängen beeinflusst den erzeugten Code, aber nicht seine Optimalität.

Die durch Huffman-Bäume beschriebenen Codes sind optimal unter denjenigen, die jedes Zeichen einzeln kodieren. Da in natürlichsprachlichen Texten einige Buchstabenkombinationen häufiger vorkommen als andere, lässt sich der Kompressionsgrad dort erhöhen, indem ganze Kombinationen von Zeichen mit einzelnen Codewörtern kodiert werden. Diese Idee liegt zum Beispiel dem ZIP- Kompressionsverfahren zugrunde.

Das in diesem Abschnitt betrachtete Kompressionsverfahren ist verlustfrei. Auch durch verlustbehaftete Kompression kann der Kompressionsgrad erhöht werden. Verlustbehaftete Kompression wird zum Beispiel zur Kodierung von Multimediadaten angewendet, worauf wir später noch eingehen werden.

Quellen und Lesetipps

- Entropie in der englischen Wikipedia
- Java-Applet zur Huffman-Kodierung samt Unterrichtsmaterial

25

Berechenbarkeit und Komplexität

Im Rahmen der Theoretischen Informatik wurden unterschiedliche Modelle entwickelt, um das intuitive Verständnis von Berechnungs-Prozessen zu formalisieren. Alan Turing hat die nach ihm benannte **Turing-Maschine** entwickelt. Sie ist keine Rechenmaschine im herkömmlichen Sinn (wie Rechenschieber oder Taschenrechner) sondern ein mathematisches Modell, dem eine Vorstellung zu Grunde liegt, wie Berechnungen ausgeführt werden (nämlich mit einem Stift auf einem unendlich langen Stück Papier). Alonzo Church hat den sogenannten **Lambda-Kalkül** entwickelt, der es erlaubt Ausdrücke aus Variablen, Funktionen und deren Anwendung auf andere Ausdrücke auszuwerten. Das Zeichen Lambda wird dabei zur Notation anonymer Funktionen verwendet und gibt dem Kalkül seinen Namen. Es hat sich herausgestellt, dass die Formalismen der Turing-Maschinen und des Lambda-Kalkül den selben Begriff von Berechnungen formalisieren (heute **Turing-Berechenbarkeit** genannt). Das heißt, jede als Turing-Maschine notierte Berechnungsvorschrift kann im Lambda-Kalkül ausgedrückt werden und umgekehrt. Turing-Berechenbarkeit umfasst also auch die Auswertung rekursiv definierter Funktionen.

In Folge dieser Beobachtung stellten Church und Turing die nach Ihnen benannte **Church-Turing-These** auf, nach der Turing-Berechenbarkeit mit dem intuitiven Berechenbarkeits-Begriff gleichzusetzen sei – dass also jede im intuitiven Sinn berechenbare mathematische Funktion mit einer Turing-Maschine oder dem Lambda-Kalkül berechenbar ist und umgekehrt. Da der intuitive Berechenbarkeits-Begriff seiner Natur nach nicht formalisiert ist, lässt sich diese These nicht formal beweisen. Dass mittlerweile viele weitere Berechnungsmodelle entwickelt wurden (von Registermaschinen über gängige Programmiersprachen bis zu Conway's Game of Life), die sich als gleichwertig zur Turing-Maschine herausgestellt haben, stützt jedoch die Church-Turing-These. Formalismen oder Programmiersprachen, die zu Turing-Maschinen äquivalent sind, nennt man **Turing-vollständig**.

In theoretischen Diskussionen wird die Chruch-Turing-These heute als wahr angenommen und auf zweierlei Weise angewendet. Eine unwensentliche aber bequeme Anwendung besteht darin, dass man häufig für Berechnungen (die intuitiv oder als Programme in einer gängigen Programmiersprache ausgedrückt sind) keine Turing-Maschine angibt und einfach annimmt, dass das mit ausreichend Gehirnschmalz und Fleiß möglich sei. Eine wesentliche Anwendung der These ergibt sich, sobald man gezeigt hat, dass es nicht möglich ist, eine gewisse Problemstellung mit einer Turing-Maschine (oder einem equivalenten Formalismus wie dem Lambda-Kalkül) zu berechnen. Aus so einem Beweis wird nämlich in der Regel gefolgert, dass es überhaupt keine (auch keine bisher nicht formalisierte) Möglichkeit gibt, das Problem zu lösen.

Solche Problemstellungen, die lösbar sind, kann man nach dem Aufwand, der dazu nötig ist, in sogenannten Komplexitätsklassen zusammenfassen. Unterschiedliche Probleme lassen sich so bezüglich ihres Aufwandes vergleichen. In der Vergangenheit haben wir bereits die O-Notation kennengelernt, mit Hilfe derer genau dies möglich ist. Später werden wir einige Komplexitätsklassen und Beziehungen dieser untereinander skizzieren.

Berechenbarkeit

Wir betrachten im folgenden sogennnte Entscheidungsprobleme, bei denen das Ergebnis der Berechnung nur *ja* oder *nein* ist, *wahr* oder *falsch*, *true* oder *false*, *1* oder *0*. Formal können solche Probleme als Sprachen (Menge von Wörtern) aufgefasst werden, die genau diejenigen Wörter enthalten, bei denen das Ergebnis *ja* ist. Ein solches Problem heißt entscheidbar (oder Turing-entscheidbar) wenn es einen in Form einer Turing-Maschine (oder in einem äquivalenten Formalismus) beschriebenen Algorithmus gibt, der für jede möglich Probleminstanz terminiert und die korrekte Antwort berechnet. Es zeigt sich, dass nicht alle Entscheidungsprobleme entscheidbar sind – ein entsprechender Algorithmus also nicht immer existiert.[1]

[1] Die Existenz unentscheidbarer Probleme folgt aus einem Kardinalitätsargument: Die Menge möglicher Algorithmen ist zwar unendlich aber interessanter Weise abzählbar (durch Gödel-Nummerierung), während die Menge aller Entscheidungsprobleme (mit Binärstrings als Eingaben) als Potenzmenge von $\{0,1\}^*$ überabzählbar und damit mächtiger ist. Es muss also Probleme geben, für die es keinen Entscheidungs-Algorithmus gibt.

Zum Beispiel hat Church gezeigt, dass es nicht möglich ist, einen Algorithmus anzugeben, der für eine beliebige mathematische Aussage entscheidet, ob sie wahr oder falsch ist. Natürlich muss man hierfür einen formalen Rahmen für erlaubte Aussagen festlegen, worauf wir an dieser Stelle aber verzichten.

Halteproblem

Ein weiteres Beispiel für ein unentscheidbares Problem ist das sogenannte Halteproblem. Dieses besteht darin, für einen be-

liebigen (als Turing-Maschine oder äquivalent beschriebenen) Algorithmus zu entscheiden, ob der Algorithmus terminiert oder nicht. Die Unentscheidbarkeit des Halteproblems bedeutet, dass es keinen Algorithmus gibt, der für einen beliebigen anderen Algorithmus entscheidet, ob jener terminiert. Natürlich ist es für bestimmte Algorithmen (und sogar bestimmte Klassen von Algorithmen) möglich zu entscheiden, ob sie terminieren. Das Halteproblem besteht jedoch darin, dies für beliebige Algorithmen in einer Turing-vollständigen Sprache zu tun.

Wir verzichten auf eine formale Diskussion des Beweises, dass das Halteproblem unentscheidbar ist. Die Beweis-Idee können wir jedoch mit Hilfe von Ruby-Programmen verdeutlichen. Der Beweis erfolgt durch Widerspruch, beginnt also mit der (letztlich widerlegten) Annahme, dass es möglich ist, das Halteproblem zu lösen.

Angenommen es gäbe einen Algorithmus, der für jeden anderen Algorithmus terminiert und korrekt antwortet, ob der gegebene Algorithmus selbst terminiert. Da Ruby Turing-vollständig ist, nehmen wir an dieser Algorithmus sei durch eine Funktion `halts?` in einer Ruby-Datei `halts.rb` beschrieben.

```
# halts.rb

def halts? prog_name
  # returns true if 'ruby prog_name' terminates
  # and false if it does not.
end
```

Als Argument erwartet diese Funktion den Dateinamen eines Ruby-Programms, in dem der zu testende Algorithmus implementiert ist. Die Funktion `halts?` kann also dieses Programm einlesen und in beliebiger Weise untersuchen, zum Beispiel auch ein- oder mehrmals mit einem Zeitlimit ausführen. Wichtig ist jedoch, dass gemäß unserer Annahme jeder Aufruf von `halts?` terminiert und richtig antwortet.

Wir können nun unsere Annahme zum Widerspruch führen, indem wir ein weiteres Programm `contradiction.rb` schreiben, dass `halts.rb` verwendet.

```
# contradiction.rb
require_relative 'halts.rb'

if halts? 'contradiction.rb' then
  while true do
  end
end
```

Dieses Programm hat die interessante Eigenschaft, dass es seinen eigenen Dateinamen als Argument an die Funktion

`halts?` übergibt. Das Ergebnis des Aufrufes wird als Bedingung in einer bedingten Anweisung verwendet, in deren Rumpf eine Endlosschleife steht. Hierdurch ergibt sich die Situation, dass `contradiction.rb` nicht terminiert, wenn `halts? 'contradiction.rb'` den Wert `true` liefert. Ferner terminiert `contradiction.rb` wenn `halts? 'contradiction.rb'` den Wert `false` liefert. Für das Programm `contradiction.rb` liefert die Funktion `halts?` also nicht die richtige Antwort. Dies steht im Widerspruch zu der Annahme, dass sie das Halteproblem löst, also auch der Annahme, dass das Halteproblem lösbar ist.

Tatsächlich ist die hier beschriebene Variante des Halteproblems eine Vereinfachung. Üblicherweise wird zur Lösung des Halteproblems ein Algorithmus gefordert, der für einen beliebigen anderen Algorithmus *und eine beliebige Eingabe* für jenen Algorithmus entscheidet, ob der gegebene Algorithmus mit der gegebenen Eingabe terminiert. Die Idee zum Beweis dieser allgemeineren Variante ist jedoch genau die selbe.

Reduzierbarkeit

Das sogenannte **Äquivalenzproblem** besteht darin zu entscheiden, ob zwei gegebene Algorithmen das gleiche Ergebnis berechnen. Auch dieses Problem ist unentscheidbar. Wir zeigen dies durch sogenannte **Reduktion** auf das (vereinfachte) Halteproblem. Dabei zeigen wir, dass wir einen Algorithmus zum Lösen des Äquivalenzproblems zur Definition eines Algorithmus zur Lösung des Halteproblems verwenden könnten. Da letzteres nicht möglich ist, können wir folgern, dass auch die Definition eines Algorithmus zur Lösung des Äquivalenzproblems nicht möglich ist.

Angenommen der Algorithmus zur Lösung des Äquivalenzproblems ist in der Datei `equivalent.rb` implementiert.

```
# equivalent.rb

def equivalent?(prog1_name, prog2_name)
  # returns true if 'ruby prog1_name'
  # prints same result as 'ruby prog2_name'
  # and false otherwise
end
```

Mit Hilfe dieses Programms können wir eine Funktio `halts?` zur Lösung des Halteproblems wie folgt definieren.

```
require_relative 'equivalent.rb'
```

```
def halts? prog_name
  File.write('prog1.rb','puts true')
  File.write(
    'prog2.rb',
    "`ruby #{prog_name}`; puts true"
  )

  return equivalent?('prog1.rb', 'prog2.rb')
end
```

Die Funktion `halts?` schreibt zwei Ruby-Programme in Dateien mit Namen `prog1.rb` und `prog2.rb` und übergibt diese Namen dann an die Funktion `equivalent?`. Die Programme `prog1.rb` und `prog2.rb` sind so definiert, dass deren Ausgaben genau dann gleich sind, wenn das in der Datei `prog_name` gespeicherte Programm terminiert. Das Programm `prog2.rb` führt dadurch dieses Programm aus und gibt anschließend `true` aus (wie `prog1.rb`). Die Ausführung geschieht mit Hilfe der schrägen Hochkommata. Die Ausgabe eines so ausgeführten Programms wird von den Hochkommata als Zeichenkette zurückgegeben aber in der obigen Definition verworfen.

Das Äquivalenzproblem ist also mindestens so schwer wie das Halteproblem, weil man zur Entscheidung des Äquivalenzproblems unter Anderem auch entscheiden muss, für welche Eingaben die übergebenen Algorithmen terminieren. Da schon das Halteproblem (zur Entscheidung, ob sie für alle Eingaben terminieren) unentscheidbar ist, gilt dies auch für das Äquivalenzproblem.

Konkretere Probleme

Im Folgenden lernen wir zwei konkretere Probleme kennen. Als erstes betrachten wir eine Art konkretes Halteproblem. Es ist bisher (Stand 2018) nicht bekannt, ob die folgende Prozedur (entsprechend der sogenannten **Collatz-Vermutung**) für alle Eingaben terminiert.

```
def collatz(n)
  while n > 1 do
    if n % 2 == 0 then
      n = n / 2
    else
      n = 3*n + 1
    end
  end
end
```

Bis heute wurde keine Eingabe gefunden, für die die Prozedur

nicht terminiert. Ein Terminierungs-Beweis für alle Eingaben steht jedoch noch aus.

Als zweites konkretes Beispiel betrachten wir das **Post'sche Korrespondenzproblem**. Bei diesem geht es um Dominosteine, die statt mit Punkten mit Worten über einem Alphabet beschriftet sind. Einen Dominostein, der oben mit `ab` und unten mit `bca` beschriftet ist, könnten wir dabei als `ab/bca` notieren. Beim Post'schen Korrespondenzproblem geht es darum, zu entscheiden, ob es möglich ist, die Dominosteine so nebeneinander zu legen, dass oben und unten das selbe Wort entsteht. Dabei (muss mindestens ein Stein und) darf jeder der Steine beliebig oft verwendet werden.

Als Beispiel betrachten wir die folgende Menge von Steinen:

```
b/ca, a/ab, ca/a, dbd/cef, abc/c, caeef/abce
```

Wenn wir diese Steine in der Folge `2, 1, 3, 2, 5` nebeneinander legen, entsteht sowohl oben als auch unten das Wort `abcaaabc`. Der vierte und der sechste Stein wird hierbei nicht verwendet. Für dieses Beispiel ist die fragliche Anordnung also möglich.

Die Unentscheidbarkeit des Post'schen Korrespondenzproblems bedeutet, dass es keinen Algorithmus gibt, der für beliebige Mengen von Dominosteinen entscheidet, ob es eine wie oben beschriebene Anordnung gibt. Der Beweis für die Unentscheidbarkeit des Post'schen Korrespondenzproblems erfolgt durch Reduktion auf das (verallgemeinerte) Halteproblem, worauf wir an dieser Stelle jedoch nicht weiter eingehen.

Eingeschränkte Berechnungsmodelle

Wir haben früher bereits erwähnt, dass jede Zählschleife durch eine bedingte Schleife ausgedrückt werden kann. Umgekehrt war dies nicht möglich, weil Zählschleifen immer terminieren, bedingte Schleifen aber nicht. Dies wirft die Frage auf, ob zumindest jedes *terminierende* Programm auch nur mit Hilfe von Zählschleifen ausgedrückt werden kann. Tatsächlich hatte David Hilbert vermutet, dass jede totale Funktion auf diese Weise definiert werden kann. Sein Schüler Wilhelm Ackermann bewies das Gegenteil.

Die nach ihm benannte Ackermann-Funktion hat zwei Argumente. $A(0,n)$ ist definiert als $n+1$; $A(m+1,0)$ ist definiert als $A(m,1)$; und $A(m+1,n+1)$ ist definiert als $A(m-1,A(m,n-1))$. Die Ackermann-Funktion ist also rekursiv. Da mit Hilfe von Variablen, Zuweisungen und bedingten Schleifen jede berechnebare Funktion ausgedrückt werden kann (entsprechende Sprachen sind Turing-vollständig), gilt dies auch für die Ackermann-

Funktion. Wenn man statt bedingter Schleifen nur noch Zählschleifen zulässt, ist dies jedoch nicht mehr möglich.

Die Eigenschaft, dass jedes Programm ohne Rekursion oder bedingte Schleifen terminiert, hat also zur Folge, dass man auch gewisse terminierende Programme nicht mehr schreiben kann. Dieses eingeschränkte Berechnungsmodell entspricht einem eingeschränkten Rekursionsbegriff, sogenannter **primitiver Rekursion**, bei der in jedem rekursiven Aufruf die Argumente *kleiner* werden (für eine angemessene Definition von *kleiner*). Die Entwicklung nicht Turing-vollständiger Programmiersprachen, in denen jedes Programm terminiert, die jedoch trotzdem die Definition möglichst vieler oder sogar aller terminierender Programme erlauben, ist Gegenstand aktueller Forschung.

Komplexität

Bisher haben wir uns mit der Unterscheidung zwischen entscheidbaren und unentscheidbaren Problemen beschäftigt. Bei Problemen, die entscheidbar sind, stellt sich die Frage, mit welchem **Aufwand** ihre Lösung verbunden ist. Mit Aufwand kann hier die **Laufzeit** gemeint sein oder der **Speicherbedarf** oder eine andere Resource. Im Folgenden beschäftigen wir uns nur mit der Ressource Zeit, also mit der Zeit-Komplexität von Algorithmen.

Schon früher haben wir mit Hilfe der **O-Notation** den zeitlichen Aufwand definierter Funktionen (z.B. zum Sortieren von Arrays) beschrieben. Dabei wird von der tatsächlichen Laufzeit abstrahiert, indem die Größenordnung der benötigten Laufzeit in Form einer Funktion angegeben wird, die die Laufzeit in Abhängigkeit der Größe der Eingabe beschränkt. Zum Beispiel haben wir gesehen, dass zum Vergleichs-basierten Sortieren mindestens $O(n \cdot log(n))$ Vergleiche benötigt werden, wenn n die Anzahl der zu sortierenden Elemente beschreibt.

Komplexitätsklassen

Zu Beginn haben wir diskutiert, dass es unterschiedliche Berechnungsmodelle (wie Turing-Maschinen oder den Lambda-Kalkül) gibt, die alle die gleiche Ausdrucksstärke haben. Bei der Ausdrucksstärke geht es jedoch nur darum, ob ein Problem überhaupt gelöst werden kann, nicht aber um den Aufwand, der dafür nötig ist. Es zeigt sich, dass alle gängigen (manche sagen alle *vernünftigen*) Berechnungsmodelle auch in Bezug auf den Aufwand in Beziehung gesetzt werden können. So unterscheide sich der Aufwand zur Lösung eines Problems in einem Modell zum Aufwand in einem anderen Modell in der

Regel nur durch ein Polynom. Dies motiviert die Definition der **Komplexitätsklasse P** für alle Probleme, die mit polynomiellen zeitlichen Aufwand gelöst werden können. Da unterschiedliche Berechnungsmodelle bezüglich des Zeit-Aufwands polynomiell in Beziehung stehen, ist die Klasse P unabhängig vom betrachteten Berechnungsmodell. Für Probleme in P gibt es also Algorithmen mit polynomiellen Aufwand, egal ob wir Algorithmen auf Turing-Maschinen, im Lambda-Kalkül oder in gängigen Programmiersprachen betrachten.

Algorithmen in der Klasse P nennt man **Effiziente Algorithmen**. Die meisten von uns betrachteten Sortier-Algorithmen sind Beispiele für effiziente Algorithmen, da ihre Laufzeit durch ein Polynom in Abhängigkeit der Eingabegröße beschränkt ist. Dieser Begriff der Effizienz ist sehr weit gefasst, da selbst Algorithmen mit einer Laufzeit in $O(n^{100})$ als effizient bezeichnet werden. In der Praxis werden manchmal bereits Algorithmen mit einer Laufzeit in $O(n^2)$ für größere Eingaben als unbrauchbar betrachtet.

Die Komplexitätsklasse **EXP** enthält alle Algorithmen, deren zeitlicher Aufwand in Abhängigkeit von n durch $O(2^{p(n)})$ (für eine Polynom-Funktion p) beschrieben wird. Der Aufwand für solche Algorithmen steigt bei wachsender Eingabe um Größenordnungen schneller als der Aufwand jedes effizienten Algorithmus. Für die Praxis sind solche Algorithmen daher häufig unbrauchbar.

Eine interessante Komplexitätsklasse zwischen P und EXP ist die Klasse **NP** für Probleme, die in einem nichtdeterministischen Berechnungsmodell mit polynomiellem zeitlichen Aufwand entschieden werden können. Gleichbedeutend umfasst die Klasse NP genau diejenigen Probleme, bei denen eine Lösung mit polynomiellen Aufwand überprüft werden kann.

Ein Beispiel für ein Problem in NP ist die Lösung von (beliebig großen) Sudoku-Puzzlen. Es ist effizient möglich *zu überprüfen*, ob eine gegebene Belegung freier Felder den Regeln von Sudoku genügt. Eine Belegung *zu finden*, die diesen Regeln genügt, ist jedoch nicht so einfach. Eine einfache Idee ist, entsprechend der Programmiertechnik *Aufzählen und Testen* alle möglichen Belegungen aufzuzählen und auf Gültigkeit zu überprüfen. *Backtracking* ist eine Optimierung dieser Idee, die schon Teillösungen verwirft, aber dadurch trotzdem nicht effizient wird.

Ein weiteres Beispiel für ein Problem in NP ist die Frage, ob eine Teilmenge einer Menge von Zahlen so gewählt werden kann, dass deren Summe einen bestimmten Wert annimmt. Zum Beispiel könnten wir fragen, ob es eine Teilmenge von $\{4, 11, 16, 21, 27\}$ gibt, deren Summe 25 ist. Die Antwort ist *ja*, denn $4 + 21 = 25$. Wir können einfach überprüfen, ob eine

gegebene Menge eine Lösung des Problems ist. Eine solche Teilmenge zu finden, scheint deutlich schwieriger zu sein. Es ist unklar, ob das effizient möglich ist.

Es ist (Stand 2018) ungeklärt, ob die Komplexitätsklassen P und NP sich unterscheiden. Da P in NP enthalten ist, geht es dabei um die Frage, ob Probleme in NP, bei denen eine Lösung effizient überprüft werden kann, auch effizient gelöst werden kann (auf eine schlauere Weise als mit *Aufzählen und Testen*). Die meisten gehen davon aus, dass dies nicht möglich ist. Ein Beweis dieser Vermutung steht jedoch noch aus.

Reduktion, Schwere und Vollständigkeit

Zum Vergleich der Komplexität von Problemen wird der Begriff der Reduktion, den wir bereits im Kontext der Entscheidbarkeit kennengelernt haben, spezialisiert. Ein Entscheidungsproblem heißt zum Beispiel **NP-schwer**, wenn jedes Entscheidungsproblem in NP auf dieses Problem **effizient reduziert** werden kann. Hierzu wird gefordert, dass jedes Problem in NP effizient in eine Instanz des Problems, auf das reduziert wird, umgewandelt werden kann und zwar so, dass die Antworten auf die Probleme übereinstimmen. Ein NP-schweres Problem ist also mindestens so schwer wie jedes Problem in NP. Interessanterweise gibt es Probleme in NP, die NP-schwer sind. Diese heißen **NP-vollständig**.

Da jedes Problem in NP auf jedes NP-vollständige Problem reduziert werden kann, hätte die Entdeckung eines effizienten Algorithmus für ein NP-vollständiges Problem zur Folge, dass die Klassen P und NP gleich wären. Damit wäre jedes Problem in NP effizient lösbar. In der Praxis wird der Nachweis von NP-Vollständigkeit eines Problems als Indiz dafür gewertet, dass es für das Problem keinen effizieten Algorithmus gibt. Sobald ein NP-vollständiges Problem gefunden ist, kann die NP-Vollständigkeit weiterer Probleme durch effiziente Reduktion bewiesen werden, denn wenn ein NP-vollständiges Problem auf ein anderes Problem in NP effizient reduziert werden kann, dann ist auch dieses Problem NP-vollständig.

Auf diese Weise wurden eine ganze Reihe von Problemen als NP-vollständig bewiesen. Zunächst wurde dabei bewiesen, dass das Erfüllbarkeitsproblem logischer Formeln (SAT) NP-vollständig ist, also jedes Problem in NP effizient auf SAT reduziert werden kann. Weitere Beweise von NP-Vollständigkeit erfolgten dann durch Reduktion von SAT (oder anderen inzwischen bekannten NP-vollständigen Probleme) auf neue Probleme. Auch die NP-Vollständigkeit des oben genannten Problem *Teilmengensumme* wird gezeigt, indem 3-SAT (eine NP-vollständige Variante des Erfüllbarkeitsproblems) auf dieses

Problem reduziert wird.

Die Begriffe Schwere und Vollständigkeit sind nicht auf die Klasse NP beschränkt. Auch in anderen Komplexitätsklassen lassen sich Schwere und Vollständigkeit durch effiziente Reduktion formalisieren.

Optimierungsprobleme und Approximation

Bisher haben wir Entscheidungsprobleme betrachtet. Lösungen sind hier Antworten der Form *ja* oder *nein*. Bei Optimierungsproblemen werden Lösungen gesucht, die eine gewisse Zielfunktion minimieren oder maximieren. Zum Beispiel könnten wir nach einer möglichst kleinen Teilmenge einer Menge von Zahlen suchen, deren Summe einen bestimmten Wert annimmt. Zu jedem Optimierungsproblem gibt es eine Entscheidungsvariante, bei der gefragt wird, ob es eine Lösung gibt, bei der die Zielfunktion durch einen bestimmten Wert beschränkt ist. (Beispiel: Gibt es eine Teilmenge aus höchstens k Zahlen, deren Summe n ist.) In der Regel steht ein Optimierungsproblem zu seinen zugehörigen Entscheidungsproblemen in Beziehung.

Ein typisches Beispiel für ein NP-vollständiges Optimierungsproblem ist das **Handlungsreisenden-Problem**, bei dem nach einer möglichst kurzen Rundreise, die eine gegebene Menge von Städten genau einmal besucht, gefragt ist. Das zugehörige Entscheidungsproblem (ob es eine Rundreise gibt, die eine gegebene Länge nicht übersteigt) ist NP-vollständig, was sich durch Reduktion auf 3-SAT zeigen lässt.

Da es für NP-schwere Probleme (vermutlich) keine effizienten Algorithmen gibt, versucht man effiziente Algorithmen zu entwickeln, die solche Probleme zwar nicht exakt aber doch möglichst gut lösen, wobei der Wert der Zielfunktion also nur geringfügig vom optimalen Wert abweicht. Was hier *möglichst gut* und *geringfügig* heißt, wollen wir an dieser Stelle nicht genauer thematisieren.

Quellen und Lesetipps

- Michael Sipser: Introduction to the Theory of Computation
- Berechenbarkeit und Komplexität an der RWTH Aachen
- Englische Wikipedia-Artikel zu Alan Turing, Alonzo Church, Church-Turing-These, Gödel-Nummerierung, Reduktion, Collatz-Vermutung, Ackermann-Funktion, P vs NP, Polynomielle Approximation

Index

Ableitung (Syntax), 64
Ableitungsbaum, 65
Abstraktion
- Benennung, 8, 37
- Datenabstraktion, 51
- Parametrisierung, 9, 37
- Wiederverwendung, 8, 37, 45

Alphabet, 63
Angewandte Informatik, *siehe* Teilgebiete
Anweisungen
- Array-Update, 52, 75
- Ausgabe, 17, 39
- Hash-Update, 79
- Prozedur-Aufrufe, 43
- Rückgabe, 38, 39
- Zuweisungen, 12, 76

Argumente, 38
Arithmetische Ausdrücke, *siehe* Ausdrücke
Array-Update, *siehe* Anweisungen, *siehe* Anweisungen
Arrays, *siehe* Ausdrücke
Attribute, 83
Aufzählen und Überprüfen, *siehe* Programmiertechniken
Ausdrücke
- Arithmetische Ausdrücke, 11
- Array-Verkettung, 52
- Array-Zugriffe, 52
- Arrays, 51
- Blöcke, 80
- Funktions-Aufrufe, 38
- Hash-Tabellen, 79
- Hash-Zugriffe, 79
- Logische Ausdrücke, 15
- Symbole, 79
- Variablen, 12, 37
- Zeichenketten, 13, 40, 45, 123

Ausführbarkeit, 7
Ausgabe, *siehe* Anweisungen, *siehe* Anweisungen
Auswertungsreihenfolge, 38

Baum, 65, 70
Bedingte Anweisungen, *siehe* Kontrollstrukturen
Bedingte Schleifen, *siehe* Kontrollstrukturen
Bedingte Verzweigungen, *siehe* Bedingte Anweisungen
Benennung, *siehe* Abstraktion
Benutzereingaben, 40
Binäre Suche, 29, 42
Binärsystem, 104
Bit, 101
Blöcke, *siehe* Ausdrücke
Blatt, 65
Bus, 104

Datenabstraktion, *siehe* Abstraktion
Datenstrukturen
- Queue, 71
- Stack, 71

De-Multiplexer, 107

Eindeutigkeit, 7
Endlichkeit, 7
Endlosschleife, *siehe* Terminierung
Euklidischer Algorithmus, 32, 85

Fehlersuche, 39
Felder, *siehe* Arrays
FIFO-Prinzip, 71
Flip-Flop, 107
For-Schleifen, *siehe* Zähl-Schleifen
Front (Baum), 65
Funktions-Aufrufe, *siehe* Ausdrücke

Gatter, 102

Generate and Test, *siehe* Aufzählen und Überprüfen
größter gemeinsamer Teiler, 25, 32, 85

Hardware-Beschreibungssprache, 102
Hash-Tabellen, *siehe* Ausdrücke
Hash-Update, *siehe* Anweisungen
Hash-Zugriffe, *siehe* Ausdrücke
Histogram, 111

If-Anweisungen, *siehe* Bedingte Anweisungen
Intervallschachtelung, 29, 42

Klassen, 83
Kontrollstrukturen
  Bedingte Anweisungen, 16
  Bedingte Schleifen, 20, 47, 53
  Schleifen, 18
  Zähl-Schleifen, 19, 46, 53
Koplexität, 35

LIFO-Prinzip, 71
Logische Ausdrücke, *siehe* Ausdrücke

Maschineninstruktion, 108
Methode, 75
Methoden, 83
Multiplexer, 107

Nichtterminalsymbol, 63

O-Notation, 96
Objekt, 75
Objekte, 83
Optionale Anweisungen, *siehe* Bedingte Anweisungen

Palindrom, 65
Parametrisierung, *siehe* Abstraktion
Pixel, 111
Postfix-Darstellung, 70, 72
Prädikat, 40
Präfix-Darstellung, 70
Praktische Informatik, *siehe* Teilgebiete
Primzahltest, 39
Programmiertechniken
  Aufzählen und Überprüfen, 25, 30
  Teilen und Herrschen, 29, 32
Programmpunkte, 22
Programmzähler, 108
Prozedur-Aufrufe, *siehe* Anweisungen
Pythagoräische Tripel, 31

Queue, *siehe* Datenstrukturen

Rückgabe, *siehe* Anweisungen
Randfälle, 29
Rastergrafik, 111
Register, 107
Rekursion
  Abbruchbedingung, 56
  Auswertung, 56

Schaltnetz, 101, 102
Schaltwerk, 101, 106
Schleifen, *siehe* Kontrollstrukturen
Schleifen-Invariante, 24, 93
Semantik, 69
Sichtbarkeit, 87
Speicher, 107
Sprache, 63, 123
Stack, *siehe* Datenstrukturen
Stackmaschine, 72
Symbole, *siehe* Ausdrücke
Syntax, 63, 69

Tabellarische Programm-Ausführung, 92
Tabellarische Programmausführung, 21
Taktsignal, 106
Technische Informatik, *siehe* Teilgebiete, *siehe* Teilgebiete
Teilen und Herrschen, *siehe* Programmiertechniken
Teilgebiete
  Angewandte Informatik, 5
  Praktische Informatik, 5
  Technische Informatik, 5, 101
  Theoretische Informatik, 5, 35, 63
Termbaum, 70
Terminalsymbol, 63
Terminierung, 7, 21
Theoretische Informatik, *siehe* Teilgebiete, *siehe* Teilgebiete
Trial and Error, *siehe* Aufzählen und Überprüfen

Variablen, *siehe* Ausdrücke
vollkommene Zahlen, 31
Von-Neumann-Architektur, 101, 108

Wahrheitswerte, *siehe* Logische Ausdrücke

While-Schleifen, *siehe* Bedingte Schleifen
Wiederverwendung, *siehe* Abstraktion

Zähl-Schleifen, *siehe* Kontrollstrukturen
Zahlenraten, 29, 42
Zeichenketten, *siehe* Ausdrücke, *siehe* Ausdrücke
Zuweisungen, *siehe* Anweisungen, *siehe* Anweisungen