# Informatik für Nebenfächler

**Priv.-Doz. Dr. Frank Huch**

Institut für Informatik, Technische Fakultät,
Christian-Albrechts-Universität zu Kiel.

Skript zur Vorlesung im Wintersemester 2013/14.

Version vom 12. Januar 2016

# Inhaltsverzeichnis


**1 Einleitung** **3**
1.1 Grundbegriffe . . . 3

**2 Programmierung** **6**
2.1 Ausdrücke . . . 6
2.2 Einfachste Algorithmen und ihre Implementierung als Ausdrücke . . . 9
2.2.1 Boolesche Werte . . . 9
2.2.2 Ganze Zahlen . . . 11
2.2.3 Gleitkommazahlen . . . 12
2.2.4 Auswertung von Ausdrücken . . . 14
2.3 Anweisungen . . . 19
2.3.1 $while$-Schleife: . . . 20
2.3.2 Größter gemeinsamer Teiler . . . 24
2.3.3 Aufzählen und Überprüfen . . . 24
2.3.4 Euklidischer Algorithmus . . . 26
2.3.5 $for$-Schleife . . . 29
2.4 Syntaxbeschreibung . . . 31
2.4.1 Präzedenzen in der BNF . . . 38
2.5 Ausdrucksstärke unterschiedlicher Statements . . . 40
2.6 Zeichenketten . . . 43
2.7 Objektorientierte Programmierung . . . 46
2.8 Prozedurale Abstraktion . . . 48
2.8.1 Funktionen . . . 48
2.8.2 Prozeduren . . . 50
2.8.3 Funktion mit Ausgabe . . . 51
2.9 Rekursion . . . 52
2.10 Objekte und ihre Identität . . . 54
2.11 Objektorientierte Datenmodellierung . . . 57
2.12 Mutierende und nicht mutierende Methoden . . . 62

**3 Datenstrukturen und Algorithmen** **65**
3.1 Arrays . . . 65
3.2 Reguläre Ausdrücke . . . 70
3.3 Datenbanken . . . 74

# 1 Einleitung

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

### Ursprünge

Mathematik
Berechnen von Folgen, Lösung von Gleichungssystemen

Elektrotechnik
Computer als Weiterentwicklung von Schaltkreisen

Nachrichtentechnik
Datenübertragung im Rechner oder im WWW

### Disziplinen der Informatik

Theoretische Informatik
Grundlagen, z.B. Komplexitätstheorie, Berechenbarkeit, Graphentheorie

Technische Informatik
Hardwarenahe Aspekte, z.B. Mikroprozessortechnik, Rechnerarchitekturen, Netzwerksysteme

Praktische Informatik
Lösen von konkreten Problemen durch Algorithmen/Programme
hierbei wichtige Aspekte: Effizienz, Softwaretechnik, Programmiersprachen, Datenbanken

## 1.1 Grundbegriffe

*Algorithmus* ist die Beschreibung einer Vorgehensweise zur Lösung von Problemen einer bestimmten Problemklasse.

*Beispiele* Kochrezept, Berechnung der Quadratwurzel, Dekodierung von DNA-Sequenzen, Berechnung von $\pi$, Verwaltungsvorschriften

**Beachte dabei:**

- in der Regel löst ein Algorithmus eine Klasse von Problemen (d.h. er ist *parametrisiert*), z.B. Quadratwurzel für beliebige Zahlen, Verwaltungsvorschrift für Hausbau. *Parameter* legt konkretes Problem der Klasse fest.

- die Vorgehensweise kann unterschiedlich detailliert formuliert werden, z.B. beim Kochrezept: „Eiweiß steif schlagen“ oder „Schüssel holen, Eiweiß und Eigelb trennen, Schneebesen nehmen, . . . “. Es ist eine Kunst, aber auch Erfahrung, den richtigen bzw. geeigneten Grad an Genauigkeit zu finden. Wichtig hierbei: von Details abstrahieren. *Abstraktion* ist eines der wichtigsten Konzepte der Informatik.

**Beispiel** Zählen von Weizenkörnern (alle gleich schwer)
Hilfsmittel: Apothekerwaage (ohne Gewichte), einen Zettel, Bleistift

**Algorithmus** Wiederhole folgende Schritte, bis du keine Körner mehr hast:

- Teile Körner in zwei gleich schwere Haufen
- Ist dies nicht möglich lege ein Korn zur Seite. Danach kannst Du sie aufteilen.
- Beschrifte den Zettel mit $\begin{cases} 1, \text{ falls du ein Korn zur Seite gelegt hast} \\ 0, \text{ sonst} \end{cases}$
- Wähle einen Haufen aus, mit dem du im nächsten Schritt weiter machst.

Drehe die Zahl auf dem Zettel um und lies das Ergebnis als Binärzahl ab. Ggf. kannst du diese Zahl in eine Dezimalzahl wandeln.

**Beispiel**

$$\left.\begin{array}{rcl} 13:2 & = & 6 \text{ R } 1 \\ 6:2 & = & 3 \text{ R } 0 \\ 3:2 & = & 1 \text{ R } 1 \\ 1:2 & = & 0 \text{ R } 1 \end{array}\right\} 1101b = (8+4+1)d = 13d$$

Algorithmen werden in der Informatik häufig mit Hilfe von Programmiersprachen *programmiert*, so dass sie auf Rechnern/Computern ausgeführt werden können. Hierbei ist zunächst keine Abstraktion mehr möglich. Ein Programm muss so konkret sein, dass ein Prozessor (winziger Befehlssatz) den Algorithmus ausführen kann.

Programmierung war zunächst nur in Maschinensprache möglich, was aber wenig komfortabel war/ist. Feste Abstraktionen der höheren *Programmiersprachen* ermöglichen elegantere Programmierung auf abstrakterem Niveau. (In dieser Vorlesung werden wir die Sprache **Ruby** kennenlernen.) Darüber hinaus sind weitere Abstraktion durch *Programmiertechniken* (Softwaretechnik) möglich.

Programmierung ist heute die Kunst/Technik ein Problem und seine Lösung in Teile zu zerlegen, so dass sich ein kompositionelles System ergibt, welches leicht zu verstehen, zu ändern und zu konfigurieren ist.

Programmiersprachen ermöglichen die Kommunikation mit dem Rechner auf möglichst „natürliche“ Weise. Programme werden entweder mit Hilfe eines *Compilers* in Maschinensprache übersetzt (z.B. *C*) oder durch ein spezielles Programm (*Interpreter*) interpretiert (z.B. *Ruby*). Mischformen sind ebenfalls möglich, bei denen ein Compiler in eine dann interpretierte Zwischensprache übersetzt (z.B. *Java*).

Bei der Programmierung unterscheidet man drei Bereiche:

- *Syntax* beschreibt die zulässigen Zeichenfolgen der Programme
- *Semantik* beschreibt, wie die Programme ausgeführt werden, also die Bedeutung der Sprachkonstrukte

- *Pragmatik* beschreibt die Idee, wie die Programmiersprache verwendet werden soll (viele Wege führen nach Rom, aber welche sind gut?). Hier spielt auch das Programmieren als Kunst/Technik hinein.

Beachte, dass bisher wenig über die Effizienz von Programmen gesagt wurde. Diese ist in der Regel *unwichtig*!
Wenige Ausnahmen: zeitkritische Algorithmen
Wichtiger meist: Verständlichkeit, Wartbarkeit, Entwicklungszeit

# 2 Programmierung

*Algorithmus:* abstrakte Beschreibung zur Lösung von Problemen

Viele Algorithmen können prinzipiell automatisch durch Computer ausgeführt werden. Hierzu muss der Algorithmus in einer konkreten Programmiersprache *implementiert* werden. Nun wollen wir ein paar grundlegende Konzepte kennenlernen, die in vielen Programmiersprachen verwendet werden und mit deren Hilfe wir einfachste Algorithmen programmieren können.

## 2.1 Ausdrücke

Aus der Mathematik kennt man Ausdrücke

$$3+4 \qquad x^2+2x+1 \qquad (x+1)^2$$

Woraus bestehen Ausdrücke?

- Basiselemente:
  - Werte (Konstanten)
    z.B. $3, 4 \in \mathbb{N}$ oder $\pi \in \mathbb{R}$
  - Variablen
    z.B. $x, y$
    repräsentieren beliebige Werte und können später mit konkreten Werten belegt werden.
- Zusammengesetzte Ausdrücke erhält man durch die Anwendung von Funktionen (genauer eigentlich Funktionssymbolen, später hierzu mehr), z.B. $+, -, \cdot$ ($*$ in der Informatik) auf bereits gebildete Ausdrücke. Eigentlich müsste man für die Eindeutigkeit bei jeder Funktionsapplikation Klammern verwenden. Oft kann man hierauf aber verzichten, da Operator-Präzedenzen berücksichtigt werden. Auch zu Präzedenzen später noch mehr.

Die meisten dieser Funktionen sind zweistellig und verknüpfen zwei Ausdrücke zu einem neuen Ausdruck: $3+4$. Zweistellige Funktionen (und manchmal auch einstellige Funktionen), die (meist mit einem Sonderzeichen) zwischen zwei Ausdrücken notiert werden, nennt man auch *Operatoren*. Beispiele für Operatoren sind $+, -$ und $\cdot$.

Was ist aber mit $\frac{\sqrt{x^2+1}}{x}$?

Auch hier finden wir mehrere Funktionsanwendungen, die aber ungewöhnlich notiert werden:

- Die zweistelligen Funktionen $+$, $\hat{}$ (Exponentiation) und $/$ (als Bruchstrich geschrieben)
- und die einstellige Funktionen $\sqrt{}$ (Wurzelfunktion).

Der Computer (die Programmiersprache) erwartet eine genormte Darstellung solcher Ausdrücke:

$$\begin{array}{rcl} x**2 & \text{statt} & x^2 \\ sqrt(x) & \text{statt} & \sqrt{x} \\ a/b & \text{statt} & \frac{a}{b} \end{array}$$

Somit können wir $\frac{\sqrt{x^2+1}}{x}$ schreiben als:

$$sqrt(x**2+1)/x$$

Entspricht dies tatsächlich dem mathematischen Term? Bei der Quadratfunktion ist es auf Grund der Notation klar, dass nur das $x$ quadriert wird. Hier verwenden wir aber die Funktion $**$. Woher wissen wir, dass in diesem Ausdruck $x$ quadriert und nicht „hoch 2+1“gerechnet wird?

Ähnlich wie in der Mathematik verwenden auch die meisten Implementierungen von Ausdrücken in Computern Präzedenzen, die festlegen, welche Operatoren stärker als andere binden. Ein Beispiel für solch eine Präzedenz ist die Punkt-vor-Strich-Rechnung: $*$ und $/$ binden stärker als $+$ und $-$. Sie ermöglicht es uns unter Umständen auf Klammern zu verzichten. Wie ist das aber mit den Operatoren $**$ und $+$?

Hierzu wäre es nützlich, unseren Ausdruck mit diesem Ausdruck:

$$sqrt(x**(2+1))/x$$

zu vergleichen. Wir untersuchen also, wie die Semantik (Bedeutung) dieser beiden Ausdrücke in der Programmiersprache Ruby ist. Zunächst beginnen wir mit einem einfacheren Beispiel: $3+4$

Um den Wert dieses Ausdrucks zu berechnen, schreiben wir unser erstes Ruby-Programm. Wir editieren (mit Hilfe eines Editors, z.B. Notepad++) eine Ruby-Datei $\mathrm{erstesProgramm.rb}$. Wichtig ist hierbei die Endung $.\mathrm{rb}$ für „Ruby-Datei“. Der Inhalt unseres ersten Programms sieht wie folgt aus:

```
puts(3+4);
```

Haltepro Der Befehl $\mathrm{puts}$ dient zur Ausgabe von Werten. Der Ausdrucken, den wir auswerten wollen, schreiben wir in die Klammer hinter den $\mathrm{puts}$-Befehl. Die Zeile beenden wir mit einem Semikolon.

Nun können wir dieses Programm, wie folgt ausführen:

- in einer Shell tippen wir: $\mathrm{ruby\ erstesProgramm.rb}$
- in Notepad++ drücken wir einfach F8
- in Atom drücken wir auf den Startbutton.

Das Programm wird ausgeführt, der Ausdruck wir ausgewertet und das Ergebnis 7 erscheint auf dem Bildschirm.

Nun wollen wir die Ergebnisse der beiden Ausdrücke $sqrt(x**2+1)/x$ und $sqrt(x**(2+1))/x$ vergleichen. Hierzu ändern wir den Ausdruck in unserem Programm:

```
puts(sqrt(x ** 2 + 1) / x);
```

Führen wir dieses Programm aus, erhalten wir folgende Ruby-Fehlermeldung:

```
erstesProgramm.rb:1:in '<main>': undefined local variable or
method 'x' for main:Object (NameError)
```

Der Grund ist, dass der Wert der Variable x unbekannt ist und Ruby den Wert des Ausdrucks nicht berechnen kann. Es gibt ja auch gar nicht „einen Wert“ für diesen Ausdruck. Wir müssen also vorher festlegen, für welchen Wert der Variablen x, wir den Ausdruck auswerten wollen. Hierzu belegen wir vorher die Variable mit einem Wert:

```
x = 42;        # Belege x mit dem Wert 42
puts(sqrt(x ** 2 + 1) / x);
```

Die Belegung einer Variablen bezeichnet man auch als *Zuweisung*. Hierauf werden wir später noch genauer eingehen.

Führen wir nun dieses Programm aus, erhalten wir erneut eine Fehlermeldung:

```
erstesProgramm.rb:2:in '<main>': undefined method 'sqrt' for
main:Object (NoMethodError)
```

Das Ruby-System kennt die Funktion sqrt nicht. Diese Funktion ist in Ruby nicht direkt verwendbar. Sie befindet sich in einem speziellen Modul Math und kann mittels Math.sqrt verwendet werden:

```
x = 42;        # Belege x mit dem Wert 42

puts(Math.sqrt(x ** 2 + 1) / x);
```

Nun können wir die unterschiedlich geklammerten Versionen unseres Ausdrucks vergleichen und sehen, dass der Operator $**$ stärker als $+$ bindet und die Klammern um den Teilausdruck $** 2$ tatsächlich nicht notwendig sind.

Der Vorteil von Präzedenzen ist, dass viele Klammern weggelssaen werden können und Ausdrücke so besser lesbar werden. Der Computer, bzw. die Programmiersprache (oder auch andere Anwendungen, wie z.B. eine Tabellenkalkulation) fügt intern automatisch die fehlenden Klammern hinzu. Die vollständig geklammerte Schreibweise für unseren Ausdruck wäre:

$$(sqrt(((x ** 2) + 1))/x)$$

Jede Operatoranwendung wird geklammert.

Neben der Präzedenz ist es auch noch wichtig, zu verstehen, wie Ausdrücke mit Operatoren mit gleicher Präzedenz geklammert werden. Als Beispiel betrachten wir den Ausdruck $5-3-2$. Wieder gibt es zwei mögliche Interpretationen dieses Terms: $5-(3-2)$ und $(5-3)-2$. Vergleichen wir wieder die unterschiedlichen Ergebnis dieser Ausdrücke, sehen wir dass der Terme $5-3-2$ dem Term $(5-3)-2$ entspricht. Es wird also links geklammert. Man sagt auch der Operator $-$ bindet *links(-assoziativ)*. Andere Operatoren können auch rechts-assoziativ binden, worauf man beim Verständnis von Ausdrücken achten sollte.

Generell gilt: wenn man sich nicht sicher ist, wie ein Operator bindet, sollten ruhig zusätzliche (überflüssige) Klammern verwendet werden.

## 2.2 Einfachste Algorithmen und ihre Implementierung als Ausdrücke

Nachdem wir Ausdrücke und ihre Implementierung in Ruby kennengelernt haben, wollen wir Ausdrücke zur Programmierung einfacher Algorithmen verwenden.

Wir betrachten folgendes Problem:

**Gegeben:** Radius $r$
**Aufgabe:** Bestimme die Fläche eines Kreises mit Radius $r$
**Ausdruck:** $\pi \cdot r^2$

Entsprechend können wir diesen Algorithmus als Ruby-Ausdruck implementieren:

```
r = 4;      # Hier legen wir den konkreten Radius fest.

puts(3.14 * (r ** 2));
```

Ausdrücke spielen in Programmiersprachen eine wichtige Rolle. Sie kommen aber auch in vielen anderen Anwendungen vor. Als weiteres Beispiel betrachten wir Tabellenkalkulationen, wie z.B. Excel, Openoffice oder Libreoffice.

Hier können Ausdrücke als Formeln verwendet werden. Anstelle von Variablen verwendet man die Zellen einer Tabelle. Adressiert werden diese durch einen Buchstaben für die Spalte, gefolgt von einer Zahl für die Zeile. Ist z.B. der Wert für den Radius in der Zelle B1 gespeichert, so können wir den Radius mit Hilfe der Formel (des Ausdrucks) $3.14*\mathrm{B1}*\mathrm{B1}$ berechnen.

Als nächstes betrachten wir ein etwas schwierigeres Problem:

**Gegeben:** Zwei Zahlen $n$ und $m$
**Aufgabe:** Bestimme das Maximum von $n$ und $m$

Diese Aufgabe können wir mit den bisherigen Funktionen nicht lösen, es sei denn, wir gehen davon aus, dass wir eine entsprechende vordefinierte Funktion zur Verfügung haben.

Ein Algorithmus zur Lösung dieses Problems sieht wie folgt aus:

Vergleiche $n$ mit $m$
Falls $n < m$ ist, dann ist $m$ das Maximum, sonst ist $n$ das Maximum.

### 2.2.1 Boolesche Werte

Welche neuen Funktionen benötigen wir, um diesen Algorithmus zu implementieren?

Zunächst den Vergleich $<$, aber was ist das Ergebnis von $n < m$?

Eine Möglichkeit:

| | |
|---|---|
| 0 | für $n$ nicht kleiner als $m$ |
| 1 | für $n$ kleiner als $m$ |

So ist dies z.B. in der Programmiersprache C realisiert. Eine bessere Lösung ist aber die Verwendung spezieller *boolescher*[1] *Werte*: false und true.

[1] Benannt nach dem englischen Mathematiker George Boole (1815-1864).

Für die Vergleichsfunktion $<$ ergibt sich dann: Der Wert false ist Ergebnis für nicht kleiner und
der Wert true ist das Ergebnis für kleiner.

Also: $3 < 4 \rightsquigarrow$ true $\quad\quad 4 < 4 \rightsquigarrow$ false .

Beachte, dass true und false zwar dem intuitiven wahr bzw. falsch entsprechen, aber dennoch Werte, wie 42 oder -15, sind. Genau wie $3 + 4$ zu 7 reduziert wird, wird $3 < 4$ zu true reduziert.

Entsprechend stehen in Ruby auch Funktionen $<=$ (für $\leq$), $>$, $>=$ (für $\geq$) und $!=$ (für $\neq$) zur Verfügung.

Nun müssen wir aber noch eine Möglichkeit finden, wie wir das „Falls ... dann ... sonst ... “ umsetzen.

Hierzu könnten wir eine Funktion $if\_then\_else$, welche drei Argumente benötigt, verwenden. Ihre Semantik ist wie folgt definiert:

**Semantik von if_then_else**

$$if\_then\_else(true, e_1, e_2) = e_1$$
$$if\_then\_else(false, e_1, e_2) = e_2$$

Beachte, dass das erste Argument ein boolescher Wert sein muss, damit diese Funktion ausgewertet werden kann. D.h. es können hier z.B. Vergleiche verwendet werden.

Damit können wir nun den Maximumsalgorithmus als Ausdruck implementieren:

$$if\_then\_else(n > m, n, m)$$

Werten wir diesen Ausdruck nun für unterschiedliche Variablenbelegungen aus, so ergibt sich:

$$\begin{array}{rcl} n = 7, m = 42 & \Rightarrow & if\_then\_else(7 > 42, 7, 42) \\ & = & if\_then\_else(False, 7, 42) \\ & = & 42 \end{array}$$

und

$$\begin{array}{rcl} n = 42, m = 8 & \Rightarrow & if\_then\_else(42 > 8, 42, 8) \\ & = & if\_then\_else(True, 42, 8) \\ & = & 42 \end{array}$$

In Ruby wird das $if\_then\_else$ nicht als Applikation einer dreistelligen Funktion notiert, sondern in einer Mixfixnotation geschrieben:

if $e_0$ then $e_1$ else $e_2$ end anstelle von $if\_then\_else(e_0, e_1, e_2)$

In der Anwendung in unserem Ausdruck für die Maximumsberechnung also

```
n = 42;
m = 7;

puts(if n > m then n else m end);
```

Die Semantik entspricht aber genau der oben skizzierten, dreistelligen Funktion $if\_then\_else$.

Auch in Tabellenkalkulationen können boolesche Ausdrücke zur Definition von Formeln verwendet werden. Hier wird das if-then-else auch tatsächlich als 3-stellige Funktion IF_THEN_ELSE

oder WENN_DANN_SONST (in der deutschen Variante) notiert. In einigen Programmen wird allerdings ein Semikolon anstelle eines Kommas zur Trennung der Argumente verwendet.

Als weiteres Beispiel betrachten wir die boolesche Funktion $OR$ (Oder).

**Gegeben:** zwei boolesche Werte in $x$ und $y$
**Gesucht:** $OR(x, y)$ mit

| $OR$ | $x = true$ | $x = false$ |
|---|---|---|
| $y = true$ | $true$ | $true$ |
| $y = false$ | $true$ | $false$ |

Mit der 3-stelligen $if\_then\_else$-Funktion können wir $OR$ als den folgenden Ausdruck definieren:

$$if\_then\_else(x,true,\\ if\_then\_else(y, true, false))$$

Der entsprechende Ruby-Ausdruck zur Berechnung von $OR$ sieht dann wie folgt aus:

```
x = true;
y = false;

puts(if x then true
          else if y then true
                    else false
               end
     end);
```

Es gibt aber auch eine noch kompaktere Definition:

```
puts(if x then true else y end);
```

In Ruby ist die Funktion $OR$ als Infixoperator || vordefiniert und kann in booleschen Ausdrücken verwendet werden. Entsprechend gibt es auch ein $AND$ (Implementierung als Übung), welches als && zur Verfügung steht.

Wir werden später noch genauer auf die korrekte Syntax und Semantik von Ausdrücken eingehen. Zunächst soll diese intuitive Idee ausreichen. Wichtig ist es insbsondere zu verstehen, dass ein Ausdruck nur unter Berücksichtigung einer konkreten Belegung aller vorkommenden Variablen ausgewertet werden kann, seine Semantik also von der aktuellen Variablenbelegung abhängt.

↓↓↓↓↓ Zusatzinhalt, der für 5 ECTSler nicht relevant ist ↓↓↓↓↓

### 2.2.2 Ganze Zahlen

Bisher habe wir Zahlen und boolesche Werte kennengelernt. Die Zahlen wollen wir noch etwas genauer untersuchen. In der Mathematik unterscheidet man unterschiedliche Zahlenmengen: natürliche Zahlen, ganze Zahlen, rationale Zahlen und reelle Zahlen. Keine dieser Zahlenmengen kann in einem Computer repräsentiert werden, da jede Menge unendlich viele

Zahlen enthält. Ein Computer besitzt aber nur einen endlichen Speicher, so dass auch nur endlich viele Zahlen dargestellt werden können.

Welche genauen Zahlendarstellungen verwendet werden, hängt von der jeweiligen Anwendung (Programmiersprache oder Tabellenkalkulation) ab. Wir wollen hier aber die wichtigsten kennenlernen.

#### Ganze Zahlen

Unersuchen wir einmal folgende Ausdrücke in Ruby:

```
puts(2**0);          # -> 1
puts(2**1);          # -> 2
puts(2**10);         # -> 1024
puts(2**100);        # -> 1267650600228229401496703205376
puts(2**1000);       # -> 10715086071862673209484250490600...
puts(0-2**1000);     # -> -10715086071862673209484250490600...
```

Alle Werte können exakt durch Ruby berechnet werden. Ruby stellt tatsächlich ganze Zahlen (fast) beliebiger Größe dar. Die Beschränkung ergibt sich nur aus dem maximal verfügbaren Speicherplatz, was aber in der Praxis so gut wie nie ein Problem darstellt.

In manchen anderen Programmiersprachen und auch vielen Prozessoren wird nur ein endlicher Zahlenbereich zur Verfügung gestellt. Hierdurch ist gewährleistet, dass alle möglichen Zahlenwerte mit einer festen Anzahl von Bits dargestellt werden können. Je nach verwendeter Bit-Länge können die folgenden Werte dargestellt werden:

| Bit-Länge | kleinste Zahl | größte Zahl |
|---|---|---|
| 8 | -128 | 127 |
| 16 | $-32.768$ | $32.767$ |
| 32 | $-2.147.483.648$ | $2.147.483.647$ |
| 64 | $-9.223.372.036.854.775.808$ | $9.223.372.036.854.775.807$ |

Falls eine Rechnung über die untere oder obere Grenze hinaus geht, gibt es in der Regel einen Überlauf und somit ein falsches Ergebnis. Z.B. würde bei 8 Bit bei der Berechnung 120+30 das Ergebnis -23 herauskommen (weitere Infos hierzu findet man unter dem Stichwort *Zweierkomplement*).

In manchen Systemen wird dieser Überlauf aber auch abgefangen und man erhält eine Fehlermeldung. In Ruby tritt dieses Problem erst gar nicht auf.

### 2.2.3 Gleitkommazahlen

Für manche praktischen Probleme ist es aber auch sinnvoll, rationale Zahlen bzw. reelle Zahlen zu betrachten. Taschenrechner und Computer bieten scheinbar auch solche Zahlen an.

In Ruby heißen diese Zahlen $\mathrm{Float}$ und man erhält sie, in dem man mindestens eine Nachkommastelle verwendet (z.B. 3.0 oder 42.75). Alle arithmetischen Operationen liefern dann auch wieder $\mathrm{Float}$-Zahlen, so dass wir gefühlt mit Dezimalbrüchen rechnen können.

Aus der Mathematik ist klar, dass wir reelle Zahlen, wie z.B. $\sqrt{2}$, nicht exakt im Computer darstellen können. Diese Zahl weist keine Periode auf und ist bereits ein unendliches Objekt,

welches nicht im endlichen Speicher des Computers dargestellt werden kann. Das Ergebnis von $\sqrt{2}$ kann also nur angenähert werden.

Entspricht Float also der Menge der rationalen Zahlen? Dies hat sich in der Praxis ebenfalls nicht bewährt, da Brüche häufig nicht gekürzt werden können und somit nach wenigen Operationen bereits eine Darstellung mit sehr großen Zählern und Nennern entsteht.

Welche Darstellung erscheint also geeignet? Eine fest begrenzte Anzahl von Vor- und/oder Nachkommastellen. Solch eine Darstellung nennt man Festkommazahl. Der Nachteil dieser Darstellung ist aber, dass bei sehr großen bzw. kleinen Werten ein recht großer Teil der Zahlendarstellung nicht für Genauigkeit genutzt werden.

Als Lösung verwendet man deshalb in der Regel *Gleitkommazahlen* (*floating point numbers*, Float). Die Idee ist, dass man sich auf Zahlen einer bestimmten Genauigkeit beschränkt. Hierbei bedeutet Genauigkeit die Anzahl der unterscheidbaren Stellen.

Für eine Genauigkeit mit vier Stellen könnne wir z.B folgende Beispielzahlen nennen:

$$1234, \quad 23,45, \quad 34560000, \quad -0,0004567, \quad 120000$$

Beachte, dass alle Zahlen auch Zahlen einer größeren Genauigkeit sein können, wie man insbesondere am Beispiel 120000 sieht, welche auch eine Zahl mit der Genauigkeit zwei Stellen ist.

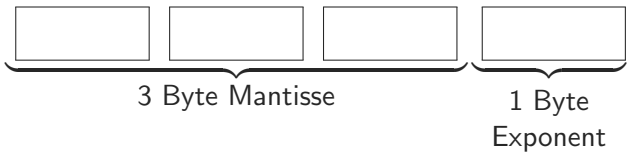
Für eine klarere Darstellung der Genauigkeit verwendet man besser eine genormte Darstellung:

$$1234 \cdot 10^0, \quad 2345 \cdot 10^{-2}, \quad 3456 \cdot 10^4, \quad -4567 \cdot 10^{-7}, 1200 \cdot 10^2$$

oder

$$0,1234 \cdot 10^4, \quad 0,2345 \cdot 10^2, \quad 0,3456 \cdot 10^8, \quad -0,4567 \cdot 10^{-3}, \quad 0,12 \cdot 10^6$$

Eine Float-Zahl wird also durch zwei Zahlen repräsentiert: die Mantisse (Ziffern der entsprechenden Genauigkeit) und einen Exponenten. Im Rechner steht für beides eine feste Anzahl von Bits (Zahlen im Binärformat) zur Verfügung. Als Beispiel sähe dies für Float (in der Regel 4 Byte) wie folgt aus:




Hiermit ergibt sich die kleinste bzw. größte darstellbare Zahl als

$$-2^{23} * 2^{127} \approx -1.42724769270596 \cdot 10^{45}$$

$$(2^{23} - 1) * 2^{127} \approx 1.4272475225647764 \cdot 10^{45}$$

Das Ruby-System versucht, Float-Werte möglichst gut für den Menschen lesbar auszugeben. Zum Beispiel beträgt die Lichtgeschwindigkeit

$$\begin{aligned} e &= 2,99792458 \cdot 10^8 \text{m/s} \\ &= 2,99792458\text{e}8\text{m/s} \end{aligned}$$

In Ruby wird dies als $299792458.0$ ausgegeben. Die Gravitationskonstante $G = 6,67384 \cdot 10^{-11} \frac{m^3}{kg \cdot s^2}$ wird aber genau in dieser Darstellung als $6.67384\text{e}{-11}$ präsentiert.

Auf Grund der Ungenauigkeiten von Float kann es zu Rundungsfehlern kommen, wie das folgende Beispiel zeigt:

```
puts(if (1-0.2-0.2-0.2-0.2-0.2)==0 then 42 else -1 end);  # -> -1
puts(1-0.2-0.2-0.2-0.2-0.2);          # -> 5.551115123125783e-17
```

Bei Abbruchbedingungen sollte man also immer prüfen, ob man bis auf eine bestimmte Nähe an den Zielwert herangekommen ist:

```
x = 1-0.2-0.2-0.2-0.2-0.2;

puts(if x >=-0.0001 || x<=0.0001 then 42 else -1 end);  # -> 42
```

Bem.: || ist der Operator für das boolesche Oder. Der Ausdruck ist erfüllt, wenn mindestens eine der beiden Vergleiche erfüllt ist.

### 2.2.4 Auswertung von Ausdrücken

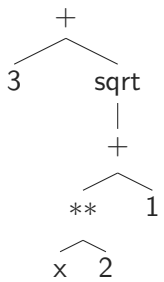
*Ausdrücke* (*Terme*) haben wir bisher so verstanden, wie sie auch in der Schulmathematik eingeführt wurden. Neu waren dabei die booleschen Werte und die dreistellige if-then-else-Funktion. Um zu verstehen, wie ein Computer mit solchen Ausdrücken umgeht, beschäftigen wir uns nun intensiver mit der Darstellung von Ausdrücken, ihrer Repräsentation im Computer und einer Auswertungsmaschine.

Ausdrücke werden (in der Mathematik) häufig auch als *Terme* bezeichnet. Durch die unterschiedliche Schachtelung der Operatoren und Funktionen bilden sie eine Baumstruktur, welche insbesondere in Form der *Termbaum*-Darstellung verdeutlicht wird.

Hierbei steht die Funktion bzw. der Operator jeweils oberhalb seiner Argumente.

**Bsp.:** $3 \ + \ sqrt(x**2 \ + \ 1)$

hat die folgende Termbaum-Repräsentation:




Die inneren Knoten des Termbaums sind mit Funktionen/Operatoren beschriftet, die Blätter mit Werten oder Variablen. Der Verzweigungsgrad der inneren Knoten ergibt sich genau aus der Stelligkeit der entsprechenden Funktion/Operation, d.h. die Kindknoten eines Funktionsknoten sind mit den Termbäumen der Argumente beschriftet. So ist die Wurzel des gesamten Termbaums mit dem Operator $+$ beschriftet. Sein linkes Kind enthält die Termbaumdarstellung des Teilterms $3$ und sein rechtes Kind die Termbaumdarstellung des Teilterms $sqrt(x**2 \ + \ 1)$.

Beachte: Es kommen keine Klammern mehr vor! Diese sind nur in der linearen Darstellung als Zeichenfolge notwendig. In der Baumstruktur werden innere/äußere Funktionsanwendung dadurch repräsentiert, dass sie weiter oben bzw. unten im Termbaum auftreten.

Gibt es noch andere Darstellungen für Terme?

Bisher: $f(g(x,y),3)$ und Infixoperatoren $(3+4)$.

Ist die Stelligkeit aller Funktionen bekannt, ist es auch möglich, die Klammern und Kommata ganz wegzulassen. Man erhält die klammerfreie Präfixnotation:

$$f\ g\ x\ y\ 3 \qquad \text{mit } f \text{ und } g \text{ 2-stellig}$$

$$\begin{aligned} +\ 3\ -\ 4\ x &\triangleq +(3,-(4,x)) \triangleq (3+(4-x)) \\ **+sqrt\ x\ 1\ /\ 4\ 2 &\triangleq **(+(sqrt(x),1),/(4,2)) \\ &\triangleq (sqrt(x)+1)**(4/2) \end{aligned}$$

Entsprechend gibt es auch die klammerfreie Postfixnotation, bei der alle Funktionsanwendungen nach den Argumenten folgen:

$$\begin{aligned} x\ y\ g\ 3\ f \\ 3\ 4\ x\ -\ + \\ x\ sqrt\ 1\ +\ 4\ 2\ /\ ** \end{aligned}$$

Wozu sind diese Termdarstellungen gut?

Insbesondere die Postfixnotation kann gut zur automatischen Auswertung des Ausdrucks verwendet werden, wie sie im Prinzip auch in vielen Implementierungen von Ausdrücken in Computern implementiert wird.

Man verwendet hierzu eine sogenannte *Stackmaschine*.

Ein *Stack* (oder Keller, Stapelspeicher) ist eine Struktur, in der beliebig viele Werte abgelegt und wieder herausgenommen werden können. Die Werte liegen hierbei „übereinander“, so dass immer nur „oben“ auf den zuletzt hinein geschriebenen Werte zugegriffen werden kann.

Es gibt nur zwei Operationen, mit denen ein Stack verändert werden kann:

**push v**, schiebt v auf den Stack

**pop**, holt oberstes Element vom Stack

**Bsp.:** Beginnen wir mit dem leeren Stack erhalten wir nach der Ausführung von push $3$:

| 3 |
|---|

Nach Ausführung von push $5$ erhalten wir:

| 5 |
|---|
| 3 |

Nach Ausführung von push $4$ erhalten wir:

| 4 |
|---|
| 5 |
| 3 |

Nach Ausführung von pop erhalten wir die 4 als Ergebnis und wieder den Stack, von vorher:

| 5 |
|---|
| 3 |

führen wir push 7 aus:

| 7 |
|---|
| 5 |
| 3 |

Mit drei weiteren pop Operationen können wir noch die Werte 7, 5 und 3 nacheinander von Stack holen.

Es werden also der Reihe nach die Werte 4, 7, 5 und 3 vom Stack ge*pop*t.

Im folgenden werden wir Stacks auch horizontal notieren. Wichtig ist aber die Beachtung des LIFO-Prinzips (**l**ast-**i**n-**f**irst-**o**ut)

---

**Einschub - FIFO-Prinzip**

Es gibt auch das FIFO-Prinzip (**f**irst-**i**n-**f**irst-**o**ut). Diese Struktur nennt man Queue (oder Schlange).

Bei obigem Beispiel hätten die pop-Operationen die Werte in der folgenden Reihenfolge geliefert:

$$3, 5, 4, 7$$

Später werden wir uns noch ausführlicher mit Queues beschäftigen.

---

Wie kann nun der Stack in der Stackmaschine verwendet werden, um den Wert eines Ausdrucks zu berechnen?
Hierzu verwendet man am besten die Postfixnotation:

**Bsp.:** $(3 + 4) * sqrt(2 * *2)$

Postfix: $3\ 4\ +\ 2\ 2 * *sqrt*$

Eine *Konfiguration der Stackmaschine* besteht jeweils aus einem Keller, neben dem der (restliche) Postfixausdruck steht. Begonnen wird mit dem leeren Keller (notieren wir als $\varepsilon$)

und der Postfixnotation des gesamten Ausdrucks:

$$
\begin{aligned}
\varepsilon \mid 3\ 4\ +\ 2\ 2\ **\ sqrt\ * \quad &\Rightarrow \quad 3 \mid 4\ +\ 2\ 2\ **\ sqrt\ * \\
&\Rightarrow \quad 3\ 4 \mid +\ 2\ 2\ **\ sqrt\ * \\
&\Rightarrow \quad 7 \mid 2\ 2\ **\ sqrt\ * \\
&\Rightarrow \quad 7\ 2 \mid 2\ **\ sqrt\ * \\
&\Rightarrow \quad 7\ 2\ 2 \mid\ **\ sqrt\ * \\
&\Rightarrow \quad 7\ 4 \mid sqrt\ * \\
&\Rightarrow \quad 7\ 2 \mid\ * \\
&\Rightarrow \quad \mathbf{14} \mid \\
&\qquad\ \uparrow \qquad\quad \nwarrow \\
&\qquad \text{Ergebnis} \quad \text{leere Eingabe}
\end{aligned}
$$

Allgemein lässt sich die Auswertung der Stackmaschine mit folgenden Regeln beschreiben:

Wir notieren eine Konfiguration der Stackmaschine mit einem $S$ für den Stack und einer Eingabe $p$ (einem Teil eines Postfixausdrucks).

Die *Startkonfiguration der Stackmaschine* besteht aus einem leeren Stack und dem auszuwertenden Term in Postfixnotation ($p_0$):

leerer Stack — initialer, zu berechnender Term in Postfixnotation

$$\varepsilon \mid p_0$$

Die *Konfigurationsübergänge der Stackmaschine* sind wie folgt definiert:

$S \mid v\ p$ mit $v$ ein Wert (z.B. Zahl, true, false)

$\Rightarrow S\ v \mid p$

$S\ v_1 \ldots v_n \mid f\ p$ mit $f$ eine $n$-stelliges Funktionssymbol und $\tilde{f}$ die Semantik von $f$

$\Rightarrow S\ \tilde{f}(v_1 \ldots v_n) \mid p$

Die *Endkonfiguration der Stackmaschine* hat die Form:

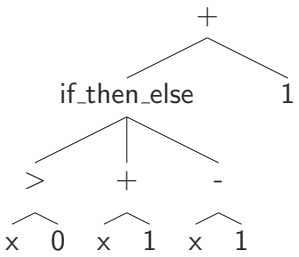
$$
\begin{aligned}
&\qquad\qquad\qquad\quad v \quad \mid \\
&\qquad\qquad\qquad\quad \uparrow \qquad \uparrow \\
&\text{nur ein! Element auf Stack} \qquad \text{leere Eingabe} \\
&\text{Das Ergebnis der Auswertung ist } v \qquad \text{(Term komplett abgearbeitet)}
\end{aligned}
$$

Beachte, dass die Argumente der Funktion eigentlich in der falschen Reihenfolge vom Stack gepopt werden. Deshalb wurde für die Termauswertung ursprünglich die Polnische Notation verwendet, in der die Argumente im Vergleich zur Postfixnotation in umgekehrter Reihenfolge angegeben werden. Wir definieren die Stackmaschine hier aber für die Postfixnotation (umgekehrte Polnische Notation) und müssen bei der Funktionsanwendung entsprechend korrekt applizieren.

**Weiteres Bsp.:**

```
if x>0 then x+1 else x-1 end +1
```

$\rightsquigarrow$ Term-Baum:




$\rightsquigarrow$ Postfix-Notation:

$$x\ 0\ >\ x\ 1\ +\ x\ 1\ -\ if\_then\_else\ 1\ +$$

Wir betrachten die Belegung: $x = 40$

Auswertung mit Stack-Maschine:

$$\begin{array}{rr|l}
 & & 40\ 0\ >\ 40\ 1\ +\ 40\ 1\ -\ if_then_else\ 1\ + \\
\Rightarrow & 40 & 0\ >\ 40\ 1\ +\ 40\ 1\ -\ if_then_else 1\ + \\
\Rightarrow & 40\ 0 & >\ 40\ 1\ +\ 40\ 1\ -\ if_then_else\ 1\ + \\
\Rightarrow & true & 40\ 1\ +\ 40\ 1\ -\ if_then_else\ 1\ + \\
\Rightarrow & true\ 40 & 1\ +\ 40\ 1\ -\ if_then_else\ 1\ + \\
\Rightarrow & true\ 40\ 1 & +\ 40\ 1\ -\ if_then_else\ 1\ + \\
\Rightarrow & true\ 41 & 40\ 1\ -\ if_then_else\ 1\ + \\
\Rightarrow^2 & true\ 41\ 40\ 1 & -\ if_then_else\ 1\ + \\
\Rightarrow & true\ 41\ 39 & if_then_else\ 1\ + \\
\Rightarrow & 41 & 1\ + \\
\Rightarrow & 41\ 1 & + \\
\Rightarrow & 42 &
\end{array}$$

Beachte im Vergleich die Auswertung als Term:

$$\begin{array}{rl}
 & if\ 40 > 0\ then\ 40 + 1\ else\ 40 - 1\ end\ +\ 1 \\
\Rightarrow & if\ true\ then\ 40 + 1\ else\ \mathbf{40 - 1}\ end\ +\ 1 \\
\Rightarrow & (40 + 1) + 1 \\
\Rightarrow & 41 + 1 \\
\Rightarrow & 42
\end{array}$$

Die Berechnung von $40 - 1$ wurde gespart.

Das $if\_then\_else$ kann auch schon ausgewertet werden, bevor das zweite und dritte Argument ausgewertet wurden, man sagt $if\_then\_else$ ist **nicht strikt im 2. und 3. Argument**.

$if\_then\_else$ ist aber auch **strikt** im 1. Argument, genau wie $+$, welches im ersten und zweiten Argument strikt ist.

Da $if\_then\_else$ entweder das zweite oder das dritte Argument liefert, ist es sinnvoll, diese vor der $if\_then\_else$-Auswertung nicht zu berechnen. Hierzu sind kompliziertere Stack-Maschinen notwendig $\rightsquigarrow$ Informatik-Studium

In der Praxis ist es aber auch sinnvoll, dass $if\_then\_else$ sein zweites und drittes Argument nicht unbedingt auswertet. So können mit Hilfe von $if\_then\_else$ auch Fehler verhindert werden:

$$if\ b == 0\ then\ 0\ else\ a/b\ end$$

verhindert Division durch Null und liefert im eigentlichen Fehlerfall das Ergebnis $0$.

↑↑↑↑↑ Zusatzinhalt, der für 5 ECTSler nicht relevant ist ↑↑↑↑↑

## 2.3 Anweisungen

Ausdrücke können keine Wiederholungen von Vorgängen ausdrücken, welche aber notwendig sind, um viele Algorithmen zu implementieren.

**Beispiel:** Fakultätsfunktion

$$n! = 1 \cdot 2 \cdot 3 \cdot \ldots \cdot n$$

**Bsp:**

$$\begin{aligned} 5! &= 1 \cdot 2 \cdot 3 \cdot 4 \cdot 5 = 120 \\ 2! &= 1 \cdot 2 = 2 \end{aligned}$$

D.h. der Ausdruck zur Fakultätsberechnung ist unterschiedlich groß für unterschiedliche Argumente $n$ (wächst mit wachsendem $n$).

Wie kann das Problem aber algorithmisch gelöst werden? Eine Lösung ist die schrittweise Multiplikation der Faktoren.

| $n$ | $n!$ |
|---|---|
| $1$ | $1$ |
| $2$ | $1 \cdot 2 = 2$ |
| $3$ | $2 \cdot 3 = 6$ |
| $4$ | $6 \cdot 4 = 24$ |
| $5$ | $24 \cdot 5 = 120$ |
| $\vdots$ | $\vdots$ |

Hierbei ist es nicht nötig sich alle vorherigen Ergebnisse zu merken. Das jeweils letzte Ergebnis reicht aus. Man kann von einem Ergebnis auf das nächste schließen:
$n! = (n-1)! \cdot n$

In Programmiersprachen können Werte in Variablen gespeichert werden, wodurch man sich in einem Programm Werte merken kann (Belegung von Variablen mit Werten). Solche Belegungen sind nun nicht mehr nur am Anfang des Programms erlaubt, sondern an jeder beliebigen Stelle. Anstelle von Variablenbelegung sagt man auch *Zuweisung*. In *imperativen Programmiersprachen* können Zuweisungen auch die existierenden Belegungen überschreiben.

**Beispiele für Zuweisungen:**

```
x = 3;
z = 4;
y = if x>z then x else z end;
y = y * 2;
```

Hierbei dürfen auf der rechten Seite der Zuweisung beliebige Ausdrücke stehen, auf der linken nur Variablen (Syntax).

Die Semantik der Zuweisung ergibt sich wie folgt: Sind alle im Ausdruck verwendeten Variablen belegt, so kann der Ausdruck zu einem Wert ausgewertet werden und die Variable wird mit diesem Wert belegt. Hierbei wird der Ausdruck stets mit der vor der Zuweisung gültigen Variablenbelegung ausgewertet. Außerdem können Zuweisungen hintereinander geschrieben und damit dann nacheinander (*sequentiell*) ausgeführt werden.

```
                <- Hier ist die Variable y noch nicht belegt.
y = 2;
                <- Hier ist y mit 2 belegt.
y = y * 2;
                <- Hier ist y mit 4 belegt.
```

Die Zeilenumbrüche sind nicht notwendig. Da jede Zuweisung mit einem Semikolon abgeschlossen wird, ist klar, wo sie endet und wo die nächste Zuweisung beginnt.

```
fac = fac * n; n = n+1;
```

Um die Semantik eines solchen Programms nachzuvollziehen, können wir die Programmausführung mit Hilfe einer Programmpunkttabelle simulieren. Hierzu geben wir jedem relevanten Programmpunkt (was zunächst alle Punkte hinter den Semikolons sind) Nummern:

```
fac = 2;          # 1
n = 3;            # 2
fac = fac * n;    # 3
n = n+1;          # 4
```

Danach erstellen wir eine Tabelle, die in der ersten Spalte die Programmpunkte protokolliert und für jede vorkommende Variable eine zusätzliche Spalte enthält. Danach werden die Zuweisungen der Reihe nach durchgeführt und die Belegung der Variable in der entsprechenden Spalte hinter dem aktuellen Programmpunkt notiert.

**Im Beispiel:**

| Programmpunkt (PP) | $fac$ | $n$ | |
|---|---|---|---|
| 1 | 2 | | ← Anfangsbelegung der Variablen $fac$ |
| 2 | | 3 | ← Anfangsbelegung der Variablen $n$ |
| 3 | 6 | | ← neue Belegungen |
| 4 | | 4 | ↙ |

Um nun nach und nach die Fakultät zu berechnen benötigen wir noch eine Wiederholungsmöglichkeit, auch *Schleife* genannt.

### 2.3.1 *while*-Schleife:

**Bsp:**

```
n = 1;          #1
while n<4 do
  n = n+1;      #2
end;            #3
```

Hierbei nennt man den Teil zwischen while und do *die Bedingung* und den Teil zwischen do und end *den Rumpf der Schleife*.

**Bedeutung:** So lange die Bedingung gilt (also zu true ausgewertet wird), wird der Rumpf wiederholt. Die Bedingung wird jedesmal vor der Ausführung des Rumpfs überprüft.

**Programmausführung:**

| PP | $n$ |
|---|---|
| 1 | 1 |
| 2 | 2 |
| 2 | 3 |
| 2 | 4 |
| 3 | |

Am Ende des Programms hat die Variable n den Wert 4. Dies können wir dem Benutzer des Programms auch noch durch die Ausgabe von n am Programmende anzeigen:

puts(n);

```
n = 1;          #1
while n<4 do
  n = n+1;      #2
end;            #3
puts(n);        #4
```

Beachte hierbei, dass in den Argumentklammern von puts auch hier ein Ausdruck steht, nämlich der Ausdruck, welcher nur aus der Variablen n besteht.

In der Programmtabelle können wir Ausgaben durch die Hinzunahme einer weiteren Spalte verdeutlichen. Das Ende der while-Schleife, sowie die puts-Anweisung vor dem 4. Programmpunkt verändern die verwendete Variable nicht.

| PP | $n$ | Ausgabe |
|---|---|---|
| 1 | 1 | |
| 2 | 2 | |
| 2 | 3 | |
| 2 | 4 | |
| 3 | | |
| 4 | | 4 |

Nun können wir die Fakultät berechnen:

```
n_max = 4;          #1   # zu berechnende Fakultaet
n = 0;              #2   # Zaehler
fac = 1;            #3   # (Teil-)Ergebnis
while n < n_max do
  n = n + 1;        #4
  fac = fac * n;    #5
end;                #6
puts(fac);          #7
```

Wieder geben wir das Ergebnis der Berechnung mit Hilfe der puts-Anweisung am Ende des Programms aus.

**Programmsimulation:** Hierzu haben wir in unserem Programm alle Semikolons durchnummeriert (im Kommentar):

| PP | $n\_max$ | $n$ | $fac$ | Ausgabe |
|---|---|---|---|---|
| 1 | 5 | | | |
| 2 | | 0 | | |
| 3 | | | 1 | |
| 4 | | 1 | | |
| 5 | | | 1 | |
| 4 | | 2 | | |
| 5 | | | 2 | |
| 4 | | 3 | | |
| 5 | | | 6 | |
| 4 | | 4 | | |
| 5 | | | 24 | |
| 6 | | | | |
| 7 | | | | 24 |

Das Ergebnis lautet also 24 und wird ausgegeben.

Zuweisungen, Sequenzen und while-Schleifen nennt man auch **Anweisungen** (Statements) und sie sind neben Ausdrücken eine weitere wichtige Struktur in imperativen Sprachen. Zunächst betrachten wir ihre Syntax nur an Beispielen. Später werden wir diese auch noch formalisieren.

Als nächstes wollen wir uns noch einmal mit dem Ausdruck zur Berechnung des Maximums beschäftigen. Zunächst hatten wir das Ergebnis dieses Ausdrucks nur ausgegeben. Nun können wir dieses Ergebnis auch in einer Variablen speichern und es so in der weiteren Berechnung verwenden:

```
n = ...;
m = ...;

max = if n>m then n else m end;
```

Die Verzweigung mittels if_then_else ist nicht nur in Ausdrücken möglich. Wir können Sie auch auf Anweisungsebene verwenden und zwischen unterschiedlichen möglichen Anweisungen Verzweigen:

```
n = ...;
m = ...;
if n >= m
  then max = n;
  else max = m;
end;
```

Die Bedingung ist auch hier weiterhin ein Ausdruck, der einen booleschen Wert als Ergebnis liefert. Der then- und der else- Teil sind nun aber beliebige Anweisungen, welche entsprechend ausgeführt werden, wenn die Bedingung zu true bzw. false ausgewertet wird.

Bei Verzweigungen auf Anweisungsebene kann es manchmal auch sinnvoll sein, den else-Fall weg zu lassen. Dies entspricht einer Verzweigung, bei der der else-Fall leer wäre. Es handelt sich also nur noch um eine so genannte $if - then$-Anweisung oder *bedingte Anweisung*, wobei der Begriff bedingte Anweisung ausdrückt, das die Anweisung(ssequenz) im then-Fall nur unter einer gegeben Bedingung ausgeführt wird.

Bei unserer Maximumsberechnung können wir die bedingte Anweisung wie folgt einsetzen:

```
n = ...;
m = ...;

max = n;
if m > max then max = m; end;
```

Will man das Maximum von drei Werten bestimmen, erkennt man den Vorteil der bedingten Anweisung noch deutlicher:

```
n = ...;
m = ...;
o = ...;

max = n;
if m > max then max = m; end;
if o > max then max = o; end;
```

Durch das schrittweise Ändern der Variablen $\max$ benötigen wir nur zwei bedingte Anweisungen, während wir in dem $if - then - else$-Ausdruck für das Maximum von drei Werten insgesamt vier (verschachtelte) $if - then - else$ Anwendungen benötigt haben.

Zum Abschluss dieses Abschnitts wollen wir noch einmal die Fakultätsberechnung betrachten. Anstatt die Faktoren hochzuzählen können wir auch runterzählen (mit Kommutativität von $\cdot$):

$$1 \cdot 2 \cdot \ldots \cdot n = n \cdot (n-1) \cdot \ldots \cdot 2 \cdot 1$$

Im Programm können wir diese Idee verwenden, um eine Variable, hier konkret $n\_max$, zu sparen:

```
n = 4; #1                          #zu berechnende Fakultaet
fac = 1; #2
while n>0 do
        fac = fac * n; #3
        n=n-1; #4
end; #5
puts(fac); #6
```

| PP | $n$ | $fac$ | Ausgabe |
|---|---|---|---|
| 1 | 4 | | |
| 2 | | 1 | |
| 3 | | 4 | |
| 4 | 3 | | |
| 3 | | 12 | |
| 4 | 2 | | |
| 3 | | 24 | |
| 4 | 1 | | |
| 3 | | 24 | |
| 4 | 0 | | |
| 5 | | | |
| 6 | | | 24 |

Beachte: Zwischenergebnisse sind keine Fakultätsergebnisse mehr. Die Eingabevariable n wird verändert $\Rightarrow$ Wert geht verloren.
Ungünstig, falls er nochmals benötigt wird!

### 2.3.2 Größter gemeinsamer Teiler

Um das Programmieren mit Schleifen weiter zu üben, wollen wir uns die Bestimmung des größten gemeinsamen Teilers zweier gegebener Zahlen anschauen. Der größte gemeinsame Teiler zweier Zahlen wird z.B. beim Kürzen von Brüchen verwendet, wo man Nenner und Zähler durch ihren größten gemeinsamen Teiler dividiert.

#### Def.: (ggT)

**Gegeben:** $a, b \in \mathbb{Z}$

**Gesucht:** $ggT(a, b) = c \in \mathbb{N}$, so dass $c$ teilt $a$ ohne Rest und $c$ 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, 21
18 hat die Teiler: 1, 2, 3, 6, 9, 18
Somit ist der größte gemeinsame Teiler: $ggt(18, 21) = 3$

0 hat die Teiler: 1, 2, 3, 4, 5, 6, 7, ...
Somit gilt für alle $a \neq 0$: $ggt(a, 0) = ggt(0, a) = a$.

$ggt(0, 0)$ ist nicht definiert, da alle Zahlen die 0 ohne Rest teilen und es somit keine größte Zahl gibt, die 0 teilt.

Wie könnte nun eine mögliche Lösung dieses Problems aussehen? Bevor wir eine geschickte Lösung mit einem etwas geschickteren Algorithmus verwenden, lernen wir eine Methode kennen, die in vielen Fällen (allerdings oft nicht besonders geschickt) zum Ziel führt.

### 2.3.3 Aufzählen und Überprüfen

Ein großer Vorteil eines Computers gegenüber einem Menschen ist die Fähigkeit, viele Werte sehr schnell aufzählen und gewisse Eigenschaften für diese Werte überprüfen 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. Dies ist auch für den ggT der Fall.

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

Für die Überprüfung, ob eine Zahl eine andere ohne Rest teilt, ist der Modulo-Operator sehr hilfreich, welcher den Rest einer ganzzahligen Division liefert. Falls $a$ und $b$ ganzzahlige Werte sind, so liefert / die ganzzahlige Division und $\%$ den Rest der ganzzahligen Division.

**Bsp.:**

$$12/9 \rightsquigarrow 1 \quad 12\%9 \rightsquigarrow 3$$
$$16/3 \rightsquigarrow 5 \quad 16\%3 \rightsquigarrow 1$$

Der Algorithmus, welcher alle möglichen Teiler aufzählt und überprüft kann wie folgt in Ruby realisiert werden:

```
a = 12;          # natuerliche Zahlen, fuer die der
b = 9;           # ggT bestimmt werden soll
test_grenze = if a<b then a else b end;
i = 1;
ggt = 1;
while i<=test_grenze do
  if a%i==0 && b%i==0 then
    ggt=i;
  end;
  i = i+1;
end;
puts(ggt);
```

Beachte, dass der Ausdruck a%i==0 && b%i==0 wegen der Präzedenzen der verwendeten Operatoren, so geklammert ist: ((a%i)==0) && ((b%i)==0), d.h. &&(logisches Und) bindet schwächer als == bindet schwächer als %.

Da wir den größten gemeinsamen Teiler suchen ist es für diese Aufgabe aber sinnvoller die Zahlen von oben nach unten aufzuzählen, da man dann bei der ersten Zahl, die beide gegebenen Zahlen ohne Rest teilt aufhören kann und den ggT ausgeben kann. Das Speichern des letzten Teilers wird überflüssig.

Es ergibt sich folgendes, einfacheres ggT-Programm:

```
a = 12;          # natuerliche Zahlen, fuer die der
b = 9;           # ggT bestimmt werden soll
ggt = if a<b then a else b end;
while a%ggt!=0 || b%ggt!=0 do
  ggt = ggt-1;
end;
puts(ggt);
```

Beachte wieder, dass der Ausdruck a%ggt!=0 || b%ggt!=0 wegen der Präzedenzen der verwendeten Operatoren, so geklammert ist: ((a%ggt)!=0) || ((b%ggt)!=0), d.h. || (logisches Oder) bindet schwächer als != bindet schwächer als %.

Problematisch sind nun noch die Randfälle $a = 0$ und/oder $b = 0$. Hier liefert das Programm einen Laufzeitfehler. Diese müssen nun noch explizit vor der Schleife abgefangen werden, was das Programm aber leider etwas aufbläht:

```
a = 12;          # natuerliche Zahlen, fuer die der
b = 9;           # ggT bestimmt werden soll
if a==0
  then
    if b==0
      then puts("ggT nicht definiert");
      else puts(b);
    end;
  else
    if b==0
      then puts(a);
      else ggt = if a<b then a else 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 der Programme auch wichtig ist, alle Randfälle systematisch zu überprüfen.

### 2.3.4 Euklidischer Algorithmus

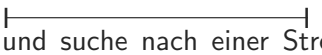
Bei großen Zahlen liefert die Aufzählen-und-Testen-Methode die Lösung leider nicht mehr in akzeptabler Zeit. Wir betrachten deshalb eine effizientere Lösung, wie sie bereits ca. 300 v. Chr. von dem griechischen Mathematiker Euklid gefunden wurde.

Gegeben zwei Strecken

Bestimme eine Strecke, mit der man beide Strecken „messen“ kann, d.h. die in beide Strecken ganz „hineinpasst“.
Beide gegebenen Strecken sollen also **Vielfache** der gesuchten Strecke sein. Hier:

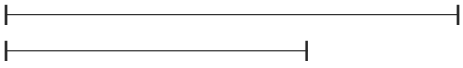
Wie findet man so eine Strecke?
Wenn beide Strecken gleich lang sind, passen sie natürlich in die jeweils andere hinein und es ist die gesuchte Strecke.
Wenn nicht: ziehe die kürzere Strecke von der Längeren ab.

Ergibt im Beispiel die Strecke:



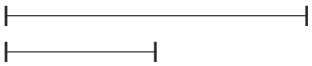
und suche nach einer Strecke, die in die kürzere Strecke und in die Strecke, die durch Abziehen der kürzeren von der längeren Strecke entsteht, hineinpasst.
Da die gesuchte Strecke sowohl in die kürzere als auch in die längere Strecke hineinpassen soll, muss sie auch in die Differenz der beiden Strecken hineinpassen.

Finde also Strecke, die in diese beiden Strecken passt:



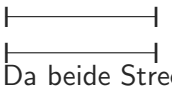
Nach Abziehen erhalten wir die Strecke:

Finde also eine Strecke, die in diese beiden Strecken passt:



Die Differenz ergibt:

Als nächstes suchen wir also die Strecke, die in die verbleibenden beiden Strecken passen:



Da beide Strecken gleich lang sind, ist dies die gesuchte Strecke.

Wir setzen das Verfahren also so lange fort, bis beide Strecken gleich lang sind. Diese Strecke ist dann die größte mögliche Strecke, die beide Strecken teilt.

Der Euklidische Algorithmus eignet sich also insbesondere auch zur Bestimmung des ggTs. Wir arbeiten ähnlich wie bei der Idee mit den Strecken, allerdings enden wir nicht wenn beide Zahlen gleich sind, sondern erst, wenn eine der beiden 0 ist. Hierdurch bestimmt unser Algorithmus auch den ggt korrekt, falls eine der beiden Zahlen 0 ist. Den Fall, dass beide Zahlen 0 sind, müssen wir dann noch separat überprüfen. **Bsp.:** $ggT(15, 10)$

$$\begin{aligned} 15 \geq 10 &\Rightarrow 15 - 10 = 5 \\ 10 \geq 5 &\Rightarrow 10 - 5 = 5 \\ 5 \geq 5 &\Rightarrow 5 - \underline{5} = 0 \end{aligned}$$

Also $ggT(15, 10) = 5$.

**Bsp.:** $ggT(12, 9)$

$$\begin{aligned} 12 \geq 9 &\Rightarrow 12 - 9 = 3 \\ 9 \geq 3 &\Rightarrow 9 - 3 = 6 \\ 6 \geq 3 &\Rightarrow 6 - 3 = 3 \\ 3 \geq 3 &\Rightarrow 3 - \underline{3} = 0 \end{aligned}$$

Also $ggT(12, 9) = 3$.

$$\begin{aligned} ggT(12, 9) &= ggT(3, 9) \\ &= ggT(3, 6) \\ &= ggT(3, 3) \\ &= ggT(0, 3) \\ &= 3 \end{aligned}$$

Wie können wir diese mathematische Definition nun in Ruby realisieren? Wir wiederholen immer wieder die gleichen Schritte, so dass wir für die Programmierung eine while-Schleife verwenden sollten. Im Rumpf der Schleife muss entweder a=a−b; oder b=b−a; gerechnet werden, je nachdem, welcher Wert größer ist. Der Schleifenrumpf sollte dann so lange wiederholt werden, solange weder a noch b Null sind:

```
a = 12; #1                    #initiale Werte
b = 9; #2
while a!=0 && b!=0 do
  if a<b
    then b=b-a; #3
```

```
    else a=a-b; #4
  end; #5
end; #6
```

Wenn wir also den Programmpunkt 6 erreichen, wissen wir, dass mindestens eine der Variablen den Wert Null hat. Die andere Variable enthält dann den ggT. Nun müssen wir nur noch herausfinden, welche Null ist. Hierbei identifizieren wir insbesondere noch den Fall, dass beide Null sind und der ggT nicht definiert ist.

```
a = 12; #1                    #initiale Werte
b = 9; #2
while a!=0 && b!=0 do
  if a<b
    then b=b-a; #3
    else a=a-b; #4
  end; #5
end; #6

if a==0
  then if b==0 then puts("ggT nicht definiert"); #7
               else puts(b); #8
       end; #9
  else puts(a); #10
end; #11
```

Die Programmausführung sieht dann wie folgt aus:

| PP | $a$ | $b$ | Ausgabe |
|---|---|---|---|
| 1 | 12 | | |
| 2 | | 9 | |
| 4 | 3 | | |
| 5 | | | |
| 3 | | 6 | |
| 5 | | | |
| 3 | | 3 | |
| 5 | | | |
| 4 | 0 | | |
| 5 | | | |
| 6 | | | |
| 8 | | | 3 |
| 9 | | | |
| 11 | | | |

Betrachte initiale Belegung: $a = 3; b = 0;$

| PP | $a$ | $b$ | Ausgabe |
|---|---|---|---|
| 1 | 3 | | |
| 2 | | 0 | |
| 6 | | | |
| 10 | | | 3 |
| 11 | | | |

Weitere sinnvolle Testfälle wären: $a = 0$, $b = 3$ und $a = 0$, $b = 0$. Außerdem können wir

auch für größere Werte die Ausgaben unserer unterschiedlichen ggT-Implementierungen vergleichen und uns so von der Korrektheit unser Programme überzeugen.

Der Algorithmus kann auch noch weiter optimiert werden, wenn man die Subtraktion durch eine Division mit Restbildung ersetzt. Näheres hierzu in der Übung.

### 2.3.5 *for*-Schleife

Bei vielen Iterationen weiß man genau, wie oft der Schleifen-Rumpf ausgeführt werden soll. Außerdem benötigt man oft eine Zählvariable, welche die Iterationen zählt und automatisch inkrementiert wird (n bei der ersten Version der Fakultät). Zu diesem Zweck kann man *for*-**Schleifen** verwenden.

**Bsp.:**

```
n = 4;                #1
fac = 1;              #2
for i in 1..n do      #3
  fac = i * fac;      #4
end;                  #5
puts(fac);            #6
```

Das Schlüsselwort for leitet die for-Schleife ein. Als nächstes wird die Zählvariable (hier i) festgelegt. Hinter dem Schlüsselwort for gibt man dann die Schleifengrenzen (Startwert und Endwert) getrennt durch zwei Punkte an. Danach folgt der Rumpf, welcher ggf. wiederholt wird. Hierbei nimmt die Zählvariable der Reihe nach die Werte vom Start- bis zum Endwert an.

Mit Hilfe der Programmpunkttabelle (da die for-Schleife die Zählvariable verändert, erhält sie auch einen Programmpunkt) können wir die Ausführung des Programms simulieren:

| PP | $n$ | $fac$ | $i$ | Ausgabe |
|---|---|---|---|---|
| 1 | 4 | | | |
| 2 | | 1 | | |
| 3 | | | 1 | |
| 4 | | 1 | | |
| 3 | | | 2 | |
| 4 | | 2 | | |
| 3 | | | 3 | |
| 4 | | 6 | | |
| 3 | | | 4 | |
| 4 | | 24 | | |
| 5 | | | | |
| 6 | | | | 24 |

Die Zählvariable wird vor jeder nächsten Iteration hochgezählt. Der letzte Durchlauf wird mit dem Endwert als Belegung für die Zählvariable durchgeführt.

Beachte: Eine *for*-**Schleife** wird immer beendet (terminiert immer), im Gegensatz zur *while*-Schleife:

**Bsp.:**

```
while true do
  ...
end;
```

terminiert nicht.

Eine Endlosschleife als Fehler ist z.B. möglich, wenn vergessen wird, die Zählvariable zu verändern:

```
while n>0 do
  fac = fac * n;    #n=n-1; vergessen
end;
```

kann die *while*-Schleife ebenfalls nicht terminieren. Dies ist bei der *for*-Schleife nicht möglich.

In Ruby terminiert die for-Schleife tatsächlich immer (außer der Rumpf terminiert nicht, z.B. weil er eine nicht-terminierende *while*-Schliefe enthält). Selbst wenn wir im Rumpf die Zählvariable verändern (was man aber auf keinen Fall machen sollte, da es zu unverständlichen Programmen führt!), wird der Wert der Zählvariable, vor dem nächsten Schleifendurchlauf auf den nächsten Zählwert gesetzt.

Dies gilt nicht für alle imperativen Programmiersprachen. Dennoch sollte man auch in anderen Sprachen die Zählvariable nicht verändern und for-Schleifen so verwenden, wie hier vorgestellt.

Genau wie bei der while-Schleife kann es aber auch bei der for-Schleife sein, dass der Rumpf gar nicht durchlaufen wird. Bei der *while*-Schleife wird die Bedingung überprüft, bevor der Rumpf ausgeführt wird. Bei der *for*-Schleife wird ebenfalls vorher überprüft, ob der Endwert bereits überschritten wurde:

```
x = 0;
for i in 5 .. 3 do
  x = x+1;
end;
puts(x);
```

gibt 0 aus.

Bei

```
for i in 5 .. 5 do
  ...
end
```

wird der Rumpf genau einmal mit $i = 5$ ausgeführt.

Nun kennen wir die beiden wichtigsten Schleifenarten: while- und for-Schleife. Beide finden sich so ähnlich in fast allen imperativen Programmiersprachen. Als Pragmatik der imperativen Programmierung gilt, dass eine for-Schleife verwendet werden sollte, wenn bei Schleifenbeginn bekannt ist, wie oft eine Wiederholung statt finden soll. Dies bedeutet nicht zwangsläufig, dass bereits zur Zeit der Programmierung die genaue Anzahl bekannt sein muss und die Schleifengrenzen immer feste Zahlen sein müssen. Es kann auch sein, dass die Grenze für die Wiederholung nur als Wert in einer Variablen bekannt ist und z.B. das Ergebnis einer anderen Berechnung oder auch einer Benutzereingabe (bekommen wir später) sein kann.

## 2.4 Syntaxbeschreibung

Bevor wir weiter in die Feinheiten der Programmierung einsteigen, wollen wir uns noch mit der Frage beschäftigen, was eigentlich genau gültige Ruby-Programme sind. Bisher haben wir die Syntax an Beispielen kennen gelernt und dann entsprechend auf andere Programme übertragen. Es stellt sich aber die Frage, ob man auch genau festlegen kann, welches die gültigen Rubyprogramme sind. Hierzu bietet die Informatik unterschiedliche Formalismen, welche es ermöglichen Sprachen formal zu definieren.

Die Ansätze gehen in der Regel auf den amerikanischen Linguisten Noam Chomsky zurück. Formal gesehen ist eine Sprache eine (möglicherweise unendliche) Menge von Wörtern.[2] Als Beispiel betrachten wir die folgenden formalen Sprachen:

- Die leere Sprache: $\emptyset$ enthält gar kein Wort.
- Eine endliche Sprachen mit genau drei Wörtern: $\{Studierende, hallo, liebe\}$
- Die Sprache, aller Wörter, die aus den Buchstaben $G$, $C$, $T$, $A$ bestehen: $\{\varepsilon$[3]$, G, C, T, A, GG, GC, GT, GA, CG, \ldots\}$. Diese Sprache ist die Sprache unseres Erbguts (der DNA). Jeder Buchstabe repräsentiert hierbei eine Nukleinbase.
- Die Sprache aller Palindrome, also der Wörter, die von vorne und hinten gelesen gleich sind: $\{\varepsilon, a, b, c, \ldots, aa, bb, cc, \ldots, aaa, aba, aca, \ldots, aaaa, abba, \ldots, aaaaa, aabaa, abcba, \ldots, otto, \ldots\}$

Zur formalen Beschreibung solcher Sprachen verwendet man in der Informatik (und auch Linguistik) unterschiedliche Formalismen: Syntaxdiagramme, Grammatiken und die Backus-Naur-Form (BNF), welche wir hier näher betrachten werden.

Die *Backus-Naur-Form (BNF)* stellt einen Formalismus zur Beschreibung von Sprachen (insbesondere Programmiersprachen) dar. Es werden Nichtterminalsymbole (beginnen mit Großbuchstaben) und Terminalsymbole (Zeichen in ' ') unterschieden. Nichtterminalsymbole stellen keine Elemente der zu definierenden Sprache dar. Vielmehr sind sie Strukturelemente, welche weiter verfeinert und letzendlich zu einer Folge von Terminalsymbolen abgeleitet werden.

Als erstes Beispiel betrachten wir eine BNF, die die Sprache des Erbguts beschreibt. Wir verwenden nur ein Nichtterminalsymbol DNA.

$$DNA \quad ::= \quad 'G'\ DNA \ \mid\ 'C'\ DNA \ \mid\ 'T'\ DNA \ \mid\ 'A'\ DNA \ \mid$$

Die möglichen Ableitungen für das Nichterminalsymbol $DNA$ werden hinter dem speziellen Symbol ::= notiert. Hierbei gibt es fünf unterschiedliche Möglichkeiten, wie wir das Nichtterminalsymbol $DNA$ ableiten können. Wir trennen die unterschiedlichen Möglichkeiten durch |. Jede einzelne Möglichkeit bezeichnen wir als *Regel* und sprechen von der Regel $DNA ::= \ 'C'\ DNA$, wenn wir eine einzelne Regel benennen. Die letzte Regel lautet $DNA ::=$.

[2] Man kann auch natürliche Sprachen, z.B. die deutsche Sprache, als formale Sprache sehen. Die Sätze würde man dann als Wörter der Sprache bezeichnen. Die einzelnen Wörter des Deutschen wären dann die Buchstaben der formalen Sprache.

[3] Die Sprache enthält insbesondere auch das leere Wort, also ein Wort, welches aus keinem Zeichen besteht. Dieses notieren wir hier als $\varepsilon$.

Nun wollen wir die spezielle Nukleinbasensequenz $CAG$ mit Hilfe dieser BNF herleiten. Dazu beginnen wir mit dem Nichtterminalsymbol, aus dem wir die Sequenz ableiten wollen. In jedem Ableitungsschritt darf jeweils nur ein vorkommendes Nichtterminalsymbol durch eine seiner rechten Seiten ersetzt werden. Kommen keine Nichtterminalsymbole mehr vor, hat man ein gültiges Wort der beschriebenen Sprache abgeleitet:

$$\begin{aligned} & DNA \\ \Rightarrow\ & C\ DNA \\ \Rightarrow\ & C\ A\ DNA \\ \Rightarrow\ & C\ A\ G\ DNA \\ \Rightarrow\ & C\ A\ G \end{aligned}$$

Im letzten Schritt verwenden wir die fünfte Regel $DNA$ ::= und ersetzen das Nichtterminalsymbol $DNA$ durch die leere Zeichenfolge, wodurch das Nichtterminalsymbol $DNA$ verschwindet und die gewünschte Zeichenfolge erzeugt wurde.

Als nächstes Beispiel betrachten wir die folgende BNF:

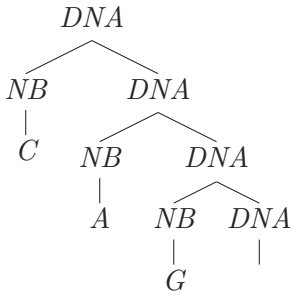
$$\begin{aligned} DNA &::= NB\ DNA \mid \\ NB &::= \ 'G' \mid 'C' \mid 'T' \mid 'A' \end{aligned}$$

Auch diese Grammatik ermöglicht es uns, alle möglichen DNA-Sequenzen herzuleiten. Als Beispiel betrachten wir noch einmal eine Ableitung der Nukleinbasensequenz $CAG$:

$$\begin{aligned} & DNA \\ \Rightarrow\ & NB\ DNA \\ \Rightarrow\ & NB\ NB\ DNA \\ \Rightarrow\ & NB\ NB\ NB\ DNA \\ \Rightarrow\ & NB\ NB\ NB \\ \Rightarrow\ & C\ NB\ NB \\ \Rightarrow\ & C\ A\ NB \\ \Rightarrow\ & C\ A\ G \end{aligned}$$

Auf Grund des zusätzlich verwendeten Nichtterminalsymbols $NB$ ist die Ableitung länger geworden. Dennoch beschreibt diese BNF die gleiche Sprache, wie die erste BNF. Wir haben über das Nichtterminalsymbol $NB$ lediglich eine weitere Struktur eingeführt, die besser verdeutlicht, dass alle vier Buchstaben Nukleinbasen repräsentieren.

Vergleichen wir die beiden Ableitungen des Wortes $CAG$ noch einmal, stellen wir fest, dass die Stelle, an der wir weiter ersetzen können, nicht mehr eindeutig definiert ist. Vielmehr gibt es mehrere mögliche Ableitungen, welche das Wort $CAG$ herleiten und auch der Schreibaufwand wächst, da immer nur ein Nichtterminalsymbol pro Schritt ersetzt werden darf. Einfacher ist es deshalb anstelle der Ableitung den Ableitungsbaum für ein gegebenes Wort hinzu schreiben:



Die inneren Knoten sind mit Nichtterminalsymbolen beschriftet. Die Blätter mit Terminalsymbolen (bzw. dem leeren Wort). Die Kinder eines Nichtterminalsymbolknotens, sind jeweils durch alle Symbole der rechten Seite der Regel definiert. Hierbei müssen Anzahl der entsprechenden Symbole und deren Reihenfolge beachtet werden. Das abgeleitete Wort erhält man, wenn man die Blätter des Ableitungsbaums von links nach rechts liest.

Die Reihenfolge, in der die Nichtterminalsymbole durch ihre rechten Seiten ersetzt werden, ist im Ableitungsbaum egal, so dass er in der Regel kompakter ist, als eine schrittweise Ableitung.

### BNF für Ruby-Werte

Als nächstes Beispiel betrachten wir eine BNF, die die Sprache aller möglichen Ruby-Werte definiert. Zunächst beschränken wir uns auf (ganze) Zahlen und die booleschen Werte als Nichtterminalsymbol $Val$:

$$
\begin{array}{rcl}
Val & ::= & Num \mid \text{'}-\text{'}\ Num \mid true \mid false \\
Num & ::= & Dig \mid Dig\ Num \\
Dig & ::= & \text{'}0\text{'} \mid \ldots \mid \text{'}9\text{'}
\end{array}
$$

Beachte, dass wir für ganze Zahlen keine führenden Nullen verbieten. Dies wäre auch möglich, soll aber, genau wie Gleitkommazahlen, in der Übung verfeinert bzw. hinzu genommen werden.

### BNF für Ruby

Nachdem wir nun Werte, wie sie in Ruby verwendet werden, definiert haben, können wir dieses Beispiel erweitern und die Sprache aller gültigen (Ruby-)Ausdrücke definieren. Hierbei beschreiben wir der Einfachheit halber nur vollständig geklammerten Ausdrücke $Exp$[4]:

$$
\begin{array}{rcl}
Exp & ::= & Var \\
 & \mid & Val \\
 & \mid & \text{'}(\text{'}\ Exp\ Op\ Exp\ \text{'})\text{'} \\
 & \mid & Fun\ \text{'}(\text{'}\ Exps\ \text{'})\text{'} \\
 & \mid & \text{'}if\text{'}\ Exp\ \text{'}then\text{'}\ Exp\ \text{'}else\text{'}\ Exp\ \text{'}end\text{'} \\
Exps & ::= & Exp \\
 & \mid & Exp\ \text{','}\ Exps \\
Op & ::= & \text{'}+\text{'} \mid \text{'}-\text{'} \mid \text{'}*\text{'} \mid \text{'}/\text{'} \mid \text{'}**\text{'} \\
Fun & ::= & \text{'}Math.sqrt\text{'} \mid \text{'}Math.sin\text{'} \mid \ldots \\
Var & ::= & \text{'}x\text{'} \mid \text{'}y\text{'} \mid \text{'}z\text{'} \mid \ldots
\end{array}
$$

[4] Wir gehen hier zunächst davon aus, dass es keine Präzedenzen gibt und die Ausdrücke vollständig geklammert werden müssen. In der Vertiefung werden wir später noch sehen, wie Präzedenzen in der BNF ausgedrückt und Klammern vermieden werden können.

Die Variablen definieren wir hier nur beispielhaft. Eine genauere Definition wird in den Übungen erfolgen.

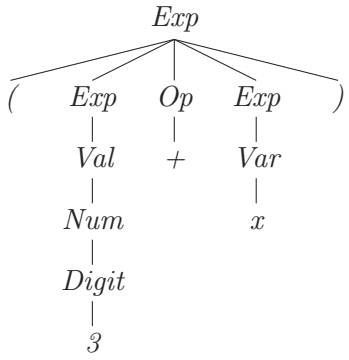
Als Beispiel für eine Ableitung in dieser BNF zeigen wir, dass das Wort $(Math.sqrt((x * * 2))/x)$ ein vollständig geklammerter Ausdruck ist, d.h. dieses Wort aus dem Nichtterminal $Exp$ abgeleitet werden kann.

Wieder darf in jedem Ableitungsschritt nur ein vorkommendes Nichtterminalsymbol durch eine seiner rechten Seiten ersetzt werden. Kommen keine Nichtterminalsymbole mehr vor, hat man ein gültiges Wort der beschriebenen Sprache abgeleitet:

$$
\begin{aligned}
& Exp \\
\Rightarrow\ & (Exp\ Op\ Exp) \\
\Rightarrow\ & (Exp\ /\ Exp) \\
\Rightarrow\ & (Exp\ /\ Var) \\
\Rightarrow\ & (Exp\ /\ x) \\
\Rightarrow\ & (Fun(Exps)\ /\ x) \\
\Rightarrow\ & (Fun(Exp)\ /\ x) \\
\Rightarrow\ & (Math.sqrt(Exp)\ /\ x) \\
\Rightarrow\ & (Math.sqrt((Exp\ Op\ Exp))\ /\ x) \\
\Rightarrow\ & (Math.sqrt((Exp * * Exp))\ /\ x) \\
\Rightarrow\ & (Math.sqrt((Exp * * Val))\ /\ x) \\
\Rightarrow\ & (Math.sqrt((Exp * * Num))\ /\ x) \\
\Rightarrow\ & (Math.sqrt((Exp * * Dig))\ /\ x) \\
\Rightarrow\ & (Math.sqrt((Exp * * 2))\ /\ x) \\
\Rightarrow\ & (Math.sqrt(((Var * * 2))\ /\ x) \\
\Rightarrow\ & (Math.sqrt((x * * 2))\ /\ x)
\end{aligned}
$$

Beachte, dass die Zeichenforlge $x * * 2$ in diesem vollständig geklammerten Ausdruck zwei mal geklammert werden muss. Eine Klammer für die Funktionsanwendung ($Math.sqrt$) und eine Klammer für die Operatoranwendung ($**$).

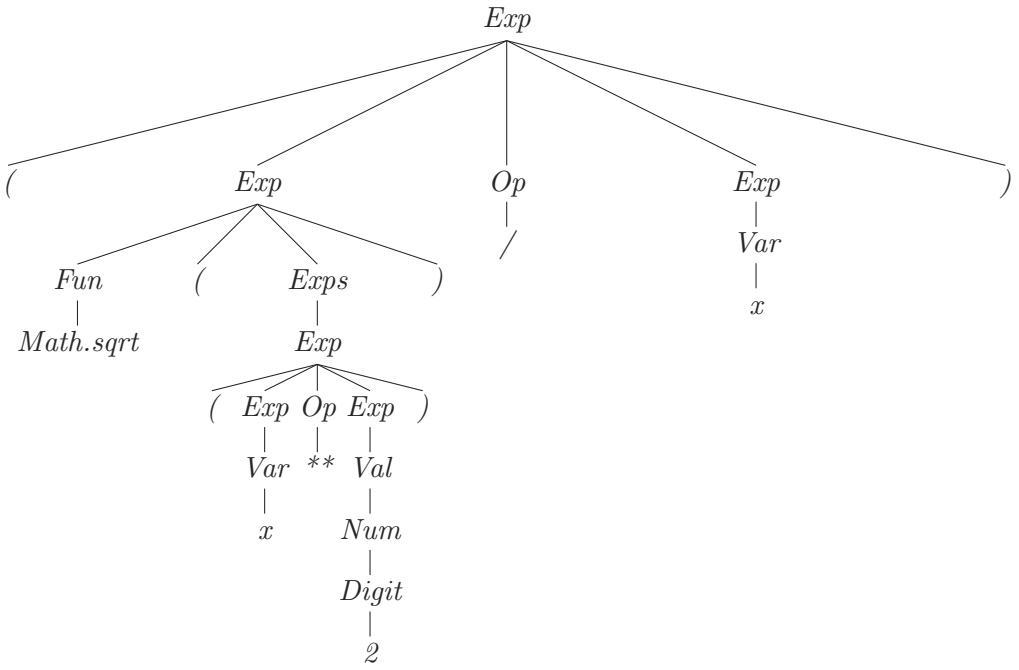
Bevor wir dieses Beispiel als Ableitungsbaum darstellen, betrachten wir den Ableitungsbaum für den Ausdruck $(3 + x)$:




Wieder sind die Nichtterminalsymbole die inneren Knoten des Ableitungsbaums. Die Wurzel ist mit dem Nichtterminal beschriftet, aus welchem man das Wort ableiten möchte (hier z.B. $Exp$). Die Kinder eines Nichtterminalknotens, entsprechen jeweils den Symbolen der rechten Seite der verwendeten Regel. So wurde bei der Wurzel zunächst die Regel $Exp\ ::=\ '('\ Exp\ Op\ Exp\ ')'$ angewendet, weshalb die Wurzel fünf Kindknoten hat, die von links nach rechts mit den entsprechenden Symbolen beschriftet sind.

Im fertigen Ableitungsbaum sind alle Blätter mit Terminalsymbolen beschriftet. Das abgeleitete Wort ergibt sich, indem man die Blätter des Baumes von links nach rechts abliest, hier also das Wort $(3 + x)$.

Der Ableitungsbaum für den Ausdruck $(Math.sqrt((x ** 2))/x)$ sieht wie folgt aus:




Mit der hier vorgestellten BNF haben wir nun schon als wichtigen Teil von Rubys Syntax Ausdrücke definiert. Im folgenden werden wir diese schrittweise erweitern und auch Anweisungen in Ruby beschreiben.

### BNF für Palindrome

Um vorher aber noch einmal klar zu machen, dass die BNF ein universeller Formalismus zur Beschreibung von Sprachen ist, wollen wir ihn vorher noch verwenden um eine Sprache zu beschreiben, die gar nichts mit Ruby zu tun hat, die Sprache der Palindrome.

Ein Palindrom ist ein Wort, welches von vorne und von hinten gelesen gleich ist. Beispiele sind **otto**, **rentner** oder (wenn man die Leer-/Satzzeichen ignoriert) **o genie, der herr ehre dein ego**. Wenn man Palindrome formal spezifizieren will, so kann man dies mit Hilfe folgender BNF machen:

$$\begin{aligned} Pal \;\; ::= \;\; & \; 'a' \; Pal \; 'a' \; \mid \; \ldots \; \mid \; 'z' \; Pal \; 'z' \\ \mid \;\; & \; 'a' \; \mid \; \ldots \; \mid \; 'z' \; \mid \end{aligned}$$

Hierbei werden natürlich nicht nur gültige Palindrome der deutschen Sprache beschrieben, sondern vielmehr alle Wörter (über dem Alphabet $'a'$ bis $'z'$), die von vorne und hinten gleich aussehen. Die letzte Regel $Pal ::=$ wird verwendet, da auch das leere Wort eine Palindrom ist. Außerdem findet sie Anwendung, falls ein Palindrom hergeleitet wird, welches keinen einzelnen Buchstaben in der Mitte hatte, wie z.B. **otto**.

Untersucht man alle BNF die wir nun programmiert haben genauer, fällt auf, dass immer wieder ähnliche Konstruktionen auftreten, wie z.B. das optionale Vorkommen oder die

Wiederholung von Teilausdrücken. Um solche Strukturen einfacher ausdrücken zu können wurden zur BNF spezielle Konstrukte hinzugefügt, was die *erweiterten BNF (EBNF)* ergibt:

- Optionales Vorkommen eines Wertes: $[\mathbf{e}]$

  Dies können wir für die Definition von Werten verwenden:

$$Val \;\; ::= \;\; [\,'-'\,]\; Num$$

- Optionale Wiederholung $\{\alpha\}$, d.h. $\alpha$ kann $0$-mal, $1$-mal, $2$-mal,... vorkommen.
- Außerdem besteht noch die Möglichkeit die Alternative (|) auch in Gruppierungen zu verwenden. Ein Beispiel hierzu ist

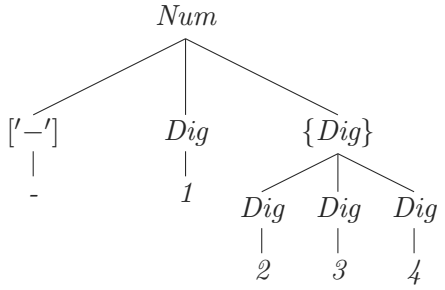
$$S \;\; ::= \;\; 'a'\; (\; 'b' \mid 'c'\;)\; d$$

  mit $abd$ und $acd$ aus $S$ ableitbar.

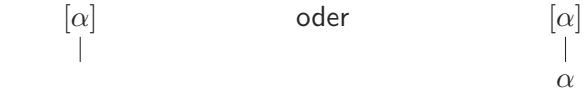
Mit diesen Abkürzungen können wir unsere BNF für Ruby-Ausdrücke an einigen Stellen kompakter aufschreiben:

$$\begin{array}{rcl} Exp & ::= & Var \\ & \mid & Val \\ & \mid & '('\; Exp\; Op\; Exp\; ')' \\ & \mid & Fun\; '('\; Exp\; \{\;,\; Exp\;\}\; ')' \\ Val & ::= & [\,'-'\,]\; Dig\; \{\; Dig\;\} \;\mid\; 'true' \;\mid\; 'false' \end{array}$$

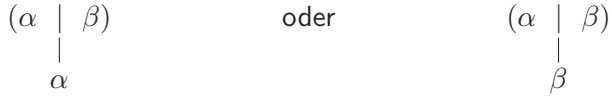
Beachte, dass sich für eine EBNF auch die Ableitungsbäume etwas ändert. Die inneren Knoten können nun auch mit den neuen Konstrukten der EBNF beschriftet sein, wobei außen immer ein neues Konstrukt steht. In einem Schritt erhält man dann so viele Kindknoten, wie benötigt werden, um dieses Konstrukt aufzulösen. Als Beispiel betrachten wir die Regel für $Num$ bei der Ableitung des Wortes -1234:




Für die anderen Regeln gilt entsprechend:




und



Hierbei stehe $\alpha$ und $\beta$ für beliebige Folgen von Terminal und Nichtterminalsymbolen und erweiterten Konstrukten der EBNF. Angewendet auf die konkreten Folgen, wie sie in der entsprechenden EBNF-Regel vorkommen.

Um nun auch die Syntax kompletter Ruby-Programme formal zu beschreiben, definieren wir noch das Nichtterminalsymbol $Stm$ für Anweisungen (*statemants*):

$$
\begin{aligned}
Stm \;::=\; & \; Stm\ Stm \\
& \mid Var\ '='\ Exp\ ';' \\
& \mid 'while'\ Exp\ 'do'\ Stm\ 'end'\ ';' \\
& \mid 'for'\ Var\ 'in'\ Exp\ '..'\ Exp\ 'do'\ Stm\ 'end'\ ';' \\
& \mid 'if'\ Exp\ 'then'\ Stm\ [\ 'else'\ Stm\ ]\ 'end'\ ';' \\
& \mid 'puts'\ '('\ Exp\ ')'\ ';'
\end{aligned}
$$

Beachte, jede Anweisung wird in der formalen Syntaxdefinition mit enem Semikolon abgeschlossen.

↓↓↓↓↓ Zusatzinhalt, der für 5 ECTSler nicht relevant ist ↓↓↓↓↓

Stellt diese Erweiterung eine echte Erweiterung dar, d.h. können Sprachen/Eigenschaften beschrieben werden, welche vorher nicht möglich waren?

Nein, denn jede EBNF kann in eine BNF übersetzt werden, welche die gleiche Sprache beschreibt. Gehe hierzu wie folgt vor:
Falls es eine Regel gibt mit

$$N \;::=\; \alpha\ [\ \beta\ ]\ \gamma$$

Dann ersetze diese durch

$$N \;::=\; \alpha\ \gamma \mid \alpha\ \beta\ \gamma$$

Falls es eine Regel gibt mit

$$N \;::=\; \alpha\ \{\ \beta\ \}\ \gamma$$

Dann ersetze diese durch

$$
\begin{aligned}
N \;&::=\; \alpha\ M\ \gamma \\
M \;&::=\; \beta\ M \;\mid\; \varepsilon,
\end{aligned}
$$

wobei $M$ ein neues Nichtterminalsymbol der EBNF ist, also noch nicht in der aktuellen EBNF verwendet worden sein darf.

Die BNF wird auch von Compilern zur Analyse der Programmiersprache verwendet, Aufgabe ist es hierbei zu einem gegebenen Wort (Programm) einen passenden Ableitungsbaum zu konstruieren. Der Compiler kann dann alle möglichen Ableitungen ausprobieren:

**Bsp:** Finde Ableitung für $(3 + 4)$

$$
\begin{aligned}
Exp \;&\Rightarrow Var\ \lightning \\
&\Rightarrow Num \Rightarrow Digit \Rightarrow Digit\ Num \;\Rightarrow 0\ \lightning \\
&\qquad\qquad\qquad\qquad\qquad\quad \Rightarrow 1\ \lightning \\
&\qquad\qquad\qquad\qquad\qquad\quad \ldots \\
&\Rightarrow (\ Exp\ Op\ Exp\ ) \Rightarrow (\ Val\ Op\ Exp\ ) \Rightarrow\ '-'\ Num \Rightarrow (\ Val\ Op\ Exp\ ) \Rightarrow \ldots
\end{aligned}
$$

Diese Suchverfahren, bei denen der Reihe nach alle möglichen Kodierungen durchgetestet werden bezeichnet man als *Backtracking*. Wir werden aus diese Programmiertechnik später noch genauer eingehen.

### 2.4.1 Präzedenzen in der BNF

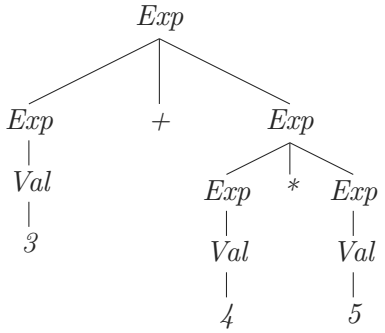
Wir wollen uns noch einmal mit der Klammerung für arithmetische Ausdrücke beschäftigen. Bisher haben wir ja in der EBNF nur vollständig geklammerte Ausdrücke betrachtet. Ruby erlaub aber auch Ausdrücke, bei denen auf Grund von Präzedenzen Klammern fehlen, z.B.:

$$3 + 4 * 5 \text{ statt } (3 + (4 * 5)) \quad \textbf{und} \quad 3 * 4 - 2 \text{ statt } ((3 * 4) - 2)$$

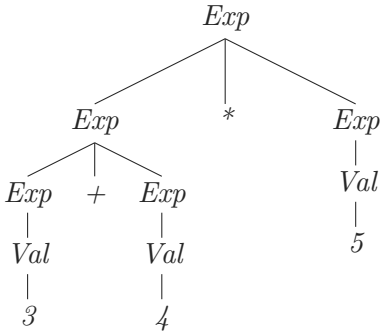
Es ist tatsächlich möglich, auch für Ausdrücke eine EBNF anzugeben, so dass die Präzedenzen berücksichtigt werden. Zur Vereinfachung, betrachten wir hier nur die beiden Operatoren + und *. Als ersten Ansatz könnten wir folgende BNF wählen:

$$\begin{aligned} Exp \;::=\; & Var \\ & \mid Val \\ & \mid Exp\ '+'\ Exp \\ & \mid Exp\ '*'\ Exp \\ & \mid '('\ Exp\ ')' \end{aligned}$$

Dann könnte der Ausdruck $3+4*5$ aber auf zwei unterschiedliche Wege abgeleitet werden:




und



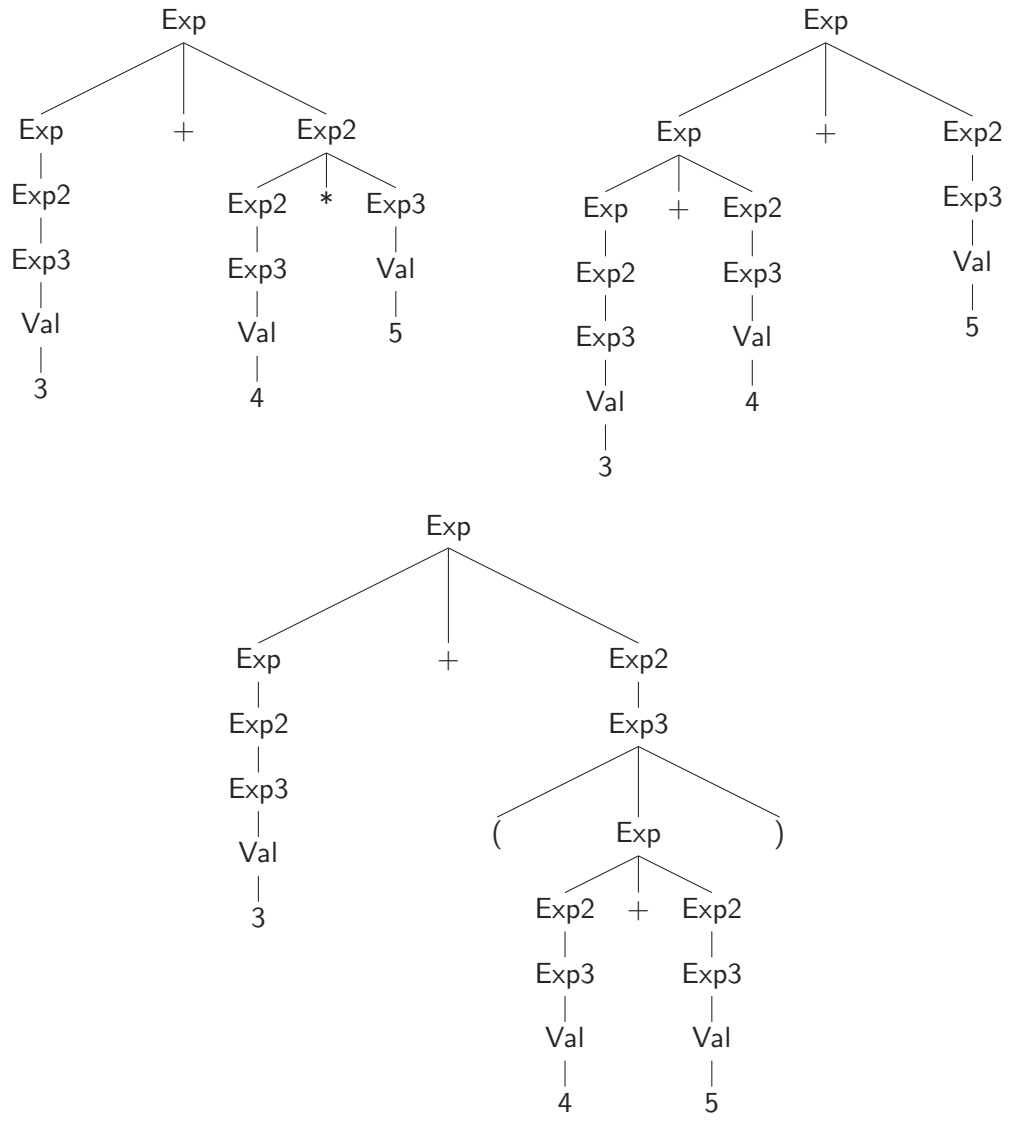

wobei der rechte Baum aber eben nicht der Strukturierung der Präzedenzen entspricht: ((3+4)*5).

Zur Realisierung von Präzedenzen können wir unterschiedliche Ebenen in der EBNF vorsehen. Hierbei sind innerhalb von Multiplikationen dann eben Additionen nur noch erlaubt, wenn diese geklammert werden. Außerdem können wir gleichzeitig noch eine Linksklammerung bei gleichen Operatoren vorsehen, d.h. 3+5+6 wird interpretiert als (3+5)+6 und eben nicht als 3+(5+6). Man sagt der Operator bindet linksassoziativ.

Die EBNF sieht dann wie folgt aus:

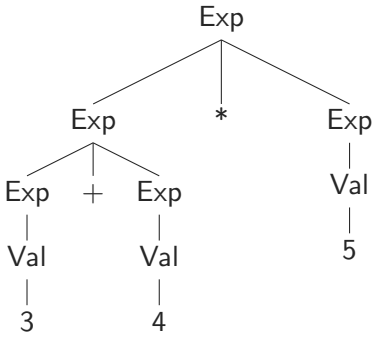
$$
\begin{aligned}
Exp &::= Exp\ '+'\ Exp2 \\
&\quad \mid Exp2 \\
Exp2 &::= Exp2\ '*'\ Exp3 \\
&\quad \mid Exp3 \\
Exp3 &::= '('\ Exp\ ')' \\
&\quad \mid Var \\
&\quad \mid Val
\end{aligned}
$$

Die zusätzlich hinzugenommen Nichtterminale *Exp2* und *Exp3* dienen dazu, zwischen beliebigen Ausdrücken ($Exp$), Ausdrücken ohne toplevel Multiplikation (*Exp2*) und Ausdrücken ohne toplevel Multiplikation und toplevel Addition zu unterscheiden. Außerdem ist bei der Regel für die Addition, eine weitere toplevel Addition nur im linken Argument erlaubt, wodurch wir die Linksassoziativität des $+$-Operators realisieren. Als Beispiele betrachten wir folgende Ableitungen:

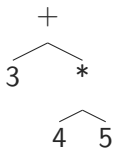


Für die explizite Rechtklammerung der Addition sind also Klammern notwendig: 3+(4+5), während die Linksklammerung auch ohne Klammern abgeleitet werden kann.

Zusammen mit Postfixnotation und Stackmaschine ergibt sich nun ein vollständiges Bild der Auswertung von Ausdrücken in Ruby. Eine gegebene Zeichenkette (z.B. 3+4*5) wird zunächst mit Hilfe einer EBNF mit Präzedenzen anlysiert. Wir erhalten den konkreten Ableitungsbaum:




Dieser wird dann in einen Termbaum (auch abstrakten Syntaxbaum genannt) umgebaut:




Dieser kann dann mittels Postfixnotation (3 4 5 * +) und Stackmaschine ausgewertet werden:

$$
\begin{aligned}
\varepsilon \mid 3\ 4\ 5\ *\ + &\Rightarrow 3 \mid 4\ 5\ *\ + \\
&\Rightarrow 3\ 4 \mid 5\ *\ + \\
&\Rightarrow 3\ 4\ 5 \mid *\ + \\
&\Rightarrow 3\ 20 \mid + \\
&\Rightarrow \mathbf{23} \mid
\end{aligned}
$$

## 2.5 Ausdrucksstärke unterschiedlicher Statements

Bisher haben wir folgende Anweisungen kennengelernt:

$$
\begin{aligned}
Stm \ ::= \ & Stm\ Stm \\
& \mid Var\ '='\ \ Exp\ ';' \\
& \mid 'while'\ Exp\ 'do'\ Stm\ 'end'\ ';' \\
& \mid 'for'\ Var\ 'in'\ Exp\ '..'\ Exp\ 'do'\ 'Stm\ 'end'\ ';' \\
& \mid 'if'\ Exp\ 'then'\ Stm\ 'else'\ Stm\ 'end'\ ';' \\
& \mid 'if'\ Exp\ 'then'\ Stm\ 'end'\ ';'
\end{aligned}
$$

Im Folgenden wollen wir uns mit der Frage beschäftigen, ob diese wirklich alle notwendig sind oder durch andere simuliert werden können. Hierbei interessieren uns insbesondere die Kontrollstruktur $while$, die beiden $if$-Varianten, sowie die *for*-Schleife.

Auf die $while$-Schleife als einzige Schleife (Wiederholungsmöglichkeit), mit der wir eine Endlosschleife programmieren können, können wir sicherlich nicht verzichten. Da die for-Schleife immer terminiert werden wir nicht alle Programme mit ihr realisieren können. Können aber die $if\_then\_else$-Anweisungen simuliert werden?

$if\_then$ ohne $else$ kann durch ein $if\_then\_else$ mit leerer $else$-Anweisung ersetzt werden. Wie konstruiert man eine leere Anweisung?

Falls eine Variable $v$ bereits verwendet wird, können wir eine Zuweisung $v = v$ als „leere“ Anweisung verwenden.

Falls noch gar keine Variable verwendet wird, können wir eine Variable $v$, welche im Programm nicht verwendet wird initialisieren $v = 0$.

Ist es umgekehrt auch möglich, das $if\_then\_else$ durch das $if\_then$ darzustellen?

Wir können einmal die Bedingung und dann die negierte Bedingung überprüfen:

```
if b then s1 else s2 end;

    ↓

if b then s1 end;
if !b then s2 end;
```

Dies ist so aber leider falsch, da sich eine Variable verändern kann, so dass $b$ einen anderen Wert hat:

```
x=5;
if x>4 then x=2; else x=7; end;
```
Endbelegung: $x = 2$

```
    ↓
x=5;
if x>4 then x=2; end;
if !(x>4) then x=7; end;
```
Endbelegung: $x = 7$

Wichtig ist es, beobachten zu können, ob $s1$ ausgeführt wurde und sonst $s2$ auszuführen: Eine mögliche Lösung ist die Verwendung eines booleschen Flags:

```
executed = false;  # neue Variable
if b then s1 executed=true; end;
if !executed then s2 end;
```

chl Alternativ können wir auch das Ergebnis der booleschen Berechnung in einer Variablen zwischenspeichern und dann zweimal im $if - then$ abfragen. Zum Speichern des Wertes der Bedingung verwenden wir eine neu, im Programm noch nicht vorkommende Variable *bool*:

```
bool = b;  # neue Variable
if bool then s1 end;
if !bool then s2 end;
```

Nun stellt sich die Frage, ob wir die $if\_then$-Anweisung auch durch eine $while$-Schleife simulieren können. Mit der vorherigen Übersetzung können wir dann auch das $if\_then\_else$ mit $while$ simulieren. ($\rightsquigarrow$ Übung)

```
if b then s end;

    ↓
```

```
while b do s end;
```

ie ausgewertet wird.

**Probleme:** $s$ wird in der Regel mehrfach ausgeführt.

**Lösung:** Wir kombnieren, diese Ideen mit der Verwendung eines booleschen Flags um mehrfache Ausführung zu verhindern:

```
bool = b;  # die Variable bool darf noch nicht verwendet sein
while bool do
  s
  bool = false;
end;
```

Als nächsten betrachten wir die Simulation von *if-then* mit Hilfe einer *for*-Schleife:

```
if b then s end;
```

↓

```
for i in 1..if b then 1 else 0 end do
  s
end
```

Hierbei haben wir die *if-then* Anweisung zwar ersetzen können. Die Verzweigung konnte allerdings nur auf Ausdrucksebene verschoben werden, da unter Verwendung der *for*-Schleife sonst keine Möglichkeit existiert, in Abhängigkeit einer booleschen Bedingung zu verzweigen.

Abschließend wollen wir noch die Simulation der *for*- mit Hilfe der *while*-Schleife betrachten. Hierbei müssen aber einige wichtige Punkte berücksichtigt werden: Veränderungen der Zählvariablen bzw. der Bereichsgrenzen haben keinerlei Auswirkungen auf das Zählverhalten der Schleife. Um dies zu realisieren, Verwenden wir eine separate Zählvariable, deren Wert unabhängig von Veränderungen der eigentlichen Zählvariable hochgezählt werden kann. Außerdem wird der Endwert der Schleife einmalig vor Schleifenbeginn berechnet und zwischen gespeichert:

```
for z in e1..e2 do
  s
end;
```

↓

```
endValue = e2;      # neue Variable
zaehler = e1;       # neue Variable
while zaehler<=endValue do
  z = x;
  s
  zaehler = zaehler+1;
end;
```

↑↑↑↑↑ Zusatzinhalt, der für 5 ECTSler nicht relevant ist ↑↑↑↑↑

## 2.6 Zeichenketten

Bei der Programmierung muss auch häufig mit Texten (Zeichenfolgen) gearbeitet werden. Hierzu stellen die meisten Programmiersprachen Zeichenketten (Strings) als Werte zur Verfügung.

In Ruby heißt die Klasse $String$ und Werte dieser Klasse können wie folgt definiert werden:

| | |
|---|---|
| "Hallo" | "Dies ist ein 'Text'." |
| 'noch ein Text' | 'ein ganz "komischer" Text' |

Strings können ebenfalls mittels $puts()$ ausgegeben werden:

$puts$("Hallo"); $\rightsquigarrow$ Hallo
$puts$('a"b"c'); $\rightsquigarrow$ a"b"c

Operationen auf Strings:

Strings können mit der Operation $+$ verbunden werden:

"Hallo" $+$ " " $+$ "Leute" $\rightsquigarrow$ "Hallo Leute"

Es gibt noch eine Reihe weiterer Funktionen für Strings. Die erste, die wir kennen lernen, dient zur Bestimmung der Länge eines Strings. Sie heißt $\mathrm{length}$. Für diese und weitere Funktionen auf Strings, gibt es aber eine Besonderheit: sie werden *Methoden* genannt und mit einem Punkt getrennt hinter ihr Argument geschrieben:

```
"Hallo".length() ⇝ 5
"Info ist toll".length() ⇝ 13
"Hi" * 3 ⇝ "HiHiHi"
```

Für die $\mathrm{length}$-Funktion schreiben wir also nicht $\mathrm{length}(\mathrm{"Hallo"})$, sondern "Hallo".length(), wobei wir den String dennoch als das Argument $\mathrm{length}$ verstehen können.

Außerdem können Strings auch mittels $\mathrm{gets}()$ vom Benutzer eingelesen werden:

**Bsp.:**

```
str = gets();
puts(str.length());
puts(str);
```

Eingabe: abc $\rightsquigarrow$ Ausgaben: $4$, abc

Warum hat $str$ die Länge 4?

Den Grund erkennt man, wenn man in $irb$ den Ausdruck gets() auswertet:

```
irb(main):001:0> gets()
abc
=> "abc\n"
```

$\backslash\mathrm{n}$ ist das Zeilenendezeichen!

Weitere spezielle Steuerzeichen in Strings:

"\t" Tabulator
"\r" Wagenrücklauf (ggf. bei Zeilenumbruch unter Windows)

Oft ist man an diesen \n nicht interessiert.

Diese können entfernt werden, mittels (Methoden):

- $strip()$ entfernt alle Whitespaces (Leerzeichen, Tabulatoren, Zeilenumbrüche) am Stringanfang und -ende.
  **Bsp.:** "␣␣\n \t \n hi \n Leute \n".strip() $\rightsquigarrow$ "hi \n Leute"
- $chop()$ entfernt pauschal das letzte Zeichen.
  **Bsp.:** "abc\n".chop $\rightsquigarrow$ "abc" "abc".chop() $\rightsquigarrow$ "ab"

Außerdem ist es möglich, auf Teilstrings zuzugreifen: $str[p, l]$, wobei $p$ die Position und $l$ die Länge des selektierten Teilstrings angiebt. Beachte hierbei aber, dass die Positionszählung bei Null beginnt, man das erste Zeichen also mit "abcdef"$[0, 1]$ und nicht mit "abcdef"$[1, 1]$ bekommt. Wir beginnen das Zählen also bei $0$:

"abcdef"$[3, 2] \rightsquigarrow$ "de"
"abcdef"$[2, 1] \rightsquigarrow$ "c"
"abcdef"$[3, 10] \rightsquigarrow$ "def"

Jetzt wollen wir mit Strings programmieren.

Aufgabe: Zähle, wie oft ein bestimmter Buchstabe in einem Text vorkommt.

```
puts("Text:");
text = gets().chop();
puts("Buchstabe:");
letter = gets()[0,1]; #wir speichern nur den ersten Buchstaben
n = 0;
for i in 0 .. text.length()-1 do
  if text[i,1] == letter
    then n = n+1;
  end;
end;
puts("Der␣Buchstabe␣'" + letter +
     "'␣kommt␣" + n.to_s() + "␣mal␣vor.");
```

Die Methode to_s() wandelt eine Zahl in einen String um (später mehr).

Als Besonderheit ist es in Ruby bei nullstelligen Methoden, also Methoden, die keinen Parameter erwarten, auch möglich, die Klammern wegzulassen, also nur n.to_s zu schreiben. Gleiches gilt auch für die String-Methode length, welche auch ohne Klammern notiert werden kann. Im Folgenden werden wir in diesen Fällen die Klammern weglassen.

Ausführen:

```
> ruby countletter.rb
Text:
Biologie ist ein schoenes Fach
Buchstabe:
i
Der Buchstabe 'i' kommt 4 mal vor.
>
```

Als nächstes Problem wollen wir untersuchen, ob ein String in einem anderen String vorkommt.

**Bsp.:**

```
text="Die Giraffe trinkt Kaffee.";
sub="affe";
i=0;
while i<text.length && text[i,sub.length]!=sub do
  i = i+1;
end;
puts("'" + sub + "' kommt in '" + text +
     "'" + if i==text.length
             then " nicht"
             else ""
           end + " vor.");
```

Um auch unsere alten Programme interaktiver gestalten zu können, ist es notwendig auch Zahlen einlesen können. Hierzu bieten Strings die Methoden to_i und to_f:

```
"123".to_i ⇝ 123\\
"-3a".to_i ⇝ -3 (Der Rest wird ignoriert)\\
"Hallo".to_i ⇝ 0\\
"-2.34".to_f ⇝ -2,34\\
"-2.34"".to_i ⇝ -2\\
```

Somit z.B. in Fakultätsprogramm:

```
n_str = gets();
n = n_str.to_i;
...
```

oder direkt:

```
n = gets().to_i;
```

Entsprechend können Zahlen mit der Methode to_s in einen String umgewandelt werden:

```
3.to_s ⇝ "3"
```

Nun wäre es schön, auch mit größeren Texten arbeiten zu können, wie z.B. einer große Textdatei.

Eine Textdatei kann mit

```
File.read(dateiname);
```

eingelesen werden. IO ist, genau wie Math eine Klasse, die auch Methoden zur Verfügung stellt.

Ersetzt man also im Programm

```
text = File.read("large.txt");
puts("Substring:");
sub = gets.chop();
...
```

So wird der Teilstring in dem Text der Datei mit dem Namen large.txt gesucht.

Abschließen wollen wir dieses Kapitel mit einem Programm abschließen, welches gegebene Strings umdreht.

```
text = gets.chop;
rev = "";     # in rev bauen wir den umgedrehten String auf
for i in 0 .. text.length-1
   rev = text[i,1]+rev;
end;

puts(rev);
```

## 2.7 Objektorientierte Programmierung

Bei unseren bisherigen Programmen haben wir zwischen Werten und Funktionen (Operationen die wir auf Werte anwenden können) unterschieden. Im letzten Kapitel haben wir aber schon gesehen, dass Ruby für manche Funktionen aber auch eine Notation als Methoden verwendet. In diesem Abschnitt wollen wir uns die Zusammenhänge etwas genauer anschauen. Wir betrachten noch einmal Ruby-Ausdrücke, wie wir sie schon kennen gelernt haben:

z.B.

$$\begin{array}{ll} 3+4 & if\ true\ then\ 42\ else\ 43\ end \\ 37\%8 & 16/3 \end{array}$$

Um die Fülle von Funktionen zu strukturieren wurde die *objektorientierte* Sicht von Daten und Funktionen auf diesen Daten erfunden.

Idee: Die Welt besteht aus **Objekten** (z.B. Bello, Waldi, Mietzi) die in **Klassen** z.B. Hund, Katze eingeteilt werden. Als weiteres Beispiel gibt es die Klasse der Zahlen, welche z.B. die Werte 42, 0 und -7 enthalten, sowie die Klasse der booleschen Werte true und false. Objekte werden dann zu einer Klasse zusammengefasst, wenn sie gleiche Eigenschaften bzw. Operationen besitzen, so können Hunde bellen, Zahlen addiert werden und boolesche Werte mit den booleschen Operationen verknüpft werden.

Operationen auf diesen Objekte bezeichnet man auch als *Methoden*.

**Bsp.:** Objekte der Klasse Hund könnten z.B. die Methoden hatRasse, hatGröße, hatGeschlecht zum Abfragen bestimmter Eigenschaften sein. Methoden, die das Verhalten von Hunden ändern könnten bellen oder spielMit sein.

Auch Zahlen haben Methoden +, -, * usw. Entsprechend besitzen boolesche Werte die Methoden & und | (Bem.: && und || sind keine Methoden. Auf die feinen Unteschiede zwischen diesen und den zugehörigen Methoden gehen wir später genauer ein.).

Um klar zu machen, dass bestimmte Funktionen/Operationen Methoden eines Objektes sind schreiben wir z.B.

$$3.+(4) \text{ statt } 3+4$$

Plus ist also eine Methode, welche für Objekte der Klasse Zahl definiert ist und als weiteren Parameter eine andere Zahl erfordert. Entsprechend würde man bei Hunden schreiben:

Die Größe des Objekts $\mathrm{Bello}$ können wir durch Zugriff auf die Methode $\mathrm{groesse}()$, welches alle Objkete der Klasse $\mathrm{Hund}$ zur Verfügung stellen wie folgt zugreifen: $\mathrm{Bello.groesse}()$.

Außerdem können wir Bello bellen lassen, indem wir die Methode $\mathrm{Bello.bellen()}$ aufrufen oder ihn mit Waldi spielen lassen: $\mathrm{Bello.spiel\_mit(Waldi)}$.

In Ruby können wir dies im Moment noch nicht mit Hunden und Katzen durchführen, aber mit den vordefinierten Datentypen:

**Bsp.:**

```
x = 3;
y = x.+(4);   ⇝ liefert Ausgabe 7.
puts(y);
```

Betrachten wir ein weiteres Bsp.

```
true + 3
```

NoMethodeError: undefined method '+' for true: TrueClass

Das heißt $true + 3$ wird übersetzt zu $true.+(3)$ und das Objekt true kennt keine Methode $+$. Entsprechend liefert $3 + true$ eine andere Fehlermeldung:
TypeError: true can't be coerced into Fixnum

Hier wird also $3.+(true)$ aufgerufen, was existiert, aber ein Argument der Klasse Fixnum (also eine Zahl) erwartet.

### Klassen

Welche Klassen (Typen) kennen wir schon?

- Zahlen:
  - Klasse Fixnum, Methoden: $+$, $-$, $*$, $/$, $\%$, $==$, $!=$, $\mathrm{to\_s}$
  - Klasse Float (Gleitkommazahlen), Methoden wie Fixnum aber nicht $==$ und $!=$ verwenden!
- boolesche Werte:
  Klasse TrueClass bzw. FalseClass,
  Methoden: $\&$ boolesches Und, $|$ boolesches Oder, $\hat{}$ exklusives Oder, außerdem ! als Präfixoperator (Negation, keine Methode der Klassen!)), $\mathrm{to\_s}$
- String:
  Methoden: $+$, $\mathrm{length}$, $\mathrm{to\_i}$, die Selektion eines Teilstrings [$pos$,$l$] und das Umdrehen eines Strings mittels $\mathrm{reverse}$

**Beispiele für Methodenaufrufe:**

| | | | | | |
|---|---|---|---|---|---|
| $3.0 + 4$ | $\rightsquigarrow$ | $3.0.+(4)$ | $\rightsquigarrow$ | $7.0$ | |
| $7/3$ | $\rightsquigarrow$ | $7./(3)$ | $\rightsquigarrow$ | $2$ | |
| $7.0/3$ | $\rightsquigarrow$ | $7.0./(3.0)$ | $\rightsquigarrow$ | $2,33333\ldots$ | |
| $7/3.0$ | $\rightsquigarrow$ | $7./(3.0)$ | $\rightsquigarrow$ | $2,33333\ldots$ | |
| $true\hat{}true$ | $\rightsquigarrow$ | $true\hat{}(true)$ | $\rightsquigarrow$ | $false$ | exklusive Oder |

Wir werden weitere Klassen (mit Methoden) kennenlernen und später auch eigene Klassen definieren.

## 2.8 Prozedurale Abstraktion

Bereits bei der Einführung des Algorithmenbegriffs haben wir folgende Aspekte diskutiert:

- Detailliertheit der Beschreibung (Verständlichkeit) (nimm Ei aus Verpackung, schlage es gegen Schüsselkante vs. trenne Ei, schlage Eiweiß)
- Wiederverwendbarkeit (Sauce Hollandaise oder Rührteig anderswo definiert)

Auch bei unseren Programmen wäre eine solche Strukturierung wünschenswert. Hierzu bieten Programmiersprachen in der Regel **Prozeduren** (ohne Rückgabewert) oder **Funktionen** (mit Rückgabewert).

### 2.8.1 Funktionen

Als Beispiel betrachten wir die Definition einer Funktion zur Berechnung der Fakultät.

```
def fac(n)
  res = 1;
  for i in 1..n do
    res = res * i;
  end;

  return res;
end;

puts(fac(5));   ⇝ 120
puts(fac(6));   ⇝ 720
```

Hierbei ist def ein Schlüsselwort, welches den Beginn einer Funktionsdefinition anzeigt und mit end abgeschlossen wird. Nach dem Schllüsselwort def steht der Name der definierten Funktion und danach kommen die formalen Parameter (Varibalen, falls mehrere durch Kommas getrennt), welche von den konkreten Parametern beim Funktionsaufruf abstrahieren. Nach den Parametern kann das Verhalten der Funktion in Form normaler Anweisungen definiert werden. Diesen Teil bezeichnet man als *Funktionsrumpf*.

Der Rückgabe-Wert der Funktionsdefinition wird mit Hilfe des Schlüsselwortes return bestimmt. Die Ausführung der return-Anweisung beendet die Funktion. Um den Code übersichtlicher zu gesatlten sollte man aber darauf achten, dass return immer am Ende einer Funktionsdefinition steht und bei Verzweigungen in allen möglichen Zweigen ein Rückgabewert mittels return festgelegt wird.

Die Fakultätsfunktion kann nach der Definition beliebig häufig im weiteren Programm verwendet werden und mit unterschiedlichen Werten aufgerufen werden.

Ein Funktionsaufruf liefert genau wie eine vordefinierte Funktionen einen Rückgabewert. Um den Ergebniswert in die weiteren Berechnungen einbauen zu können, werden Funktionen im Rahmen von Ausdrücken aufgerufen, z.B.

```
x = fac(4);
if fac(5)>100 then
  puts("gross");
else
```

```
  puts("klein");
end;
puts("Fakultaet von " + n.to_s + " lautet " + fac(n).to_s);
```

Beim Aufruf einer definierten Funktion, werden zunächst die Argumente ausgewertet. Danach werden die formalen Parameter (Variablen der Funktionsdefinition) an die aktuellen Parameter (Argumente) gebunden. Mit dieser Bindung wird der Funktionsrumpf ausgewertet und, nach Beendigung dieser Auswertung, der Aufruf durch den Rückgabewert (Wert hinter return) ersetzt. Im Beispiel würde im Ausdruck $\mathrm{fac}(\mathrm{fac}(3))+10$ zunächst $\mathrm{fac}(3)$ berechnet, wozu mit der Belegung $\mathrm{n}{=}3$ die Fakultätsfunktion ein erstes Mal ausgeführt wird. Die Variable res wird für die Belegung $\mathrm{n}{=}3$ beim return an $6$ gebunden sein und der Aufruf zu $\mathrm{fac}(6)+10$ ausgewertet werden. Danach erfolgt der nächste Aufruf mit der Belegung $\mathrm{n}{=}6$. Disesmal erhalten wir die finale Belegung $\mathrm{res}{=}720$ und die Berechnung wird mit $720{+}10$ fortgesetzt, was abschließend noch zu $730$ ausgerechnet wird.

Bei der Definition eigener Funktionen muss man beachten, dass im Funktionsrumpf nur die Parametervariablen bekannt sind. D.h. andere Varibalen, auch die, die vor der Funktion definiert wurden, sind nicht bekannt, wie folgendes Beispiel zeigt:

```
x = 42;

def bla()
  if x == 42
    then return(42);
    else return(0);
  end;
end;

puts(bla());
```

Bei der Ausführung dieses Programms wird weder $42$ noch $0$ ausgegeben. Vielmehr erhält man folgende Fehlermeldung:

```
func.rb:4:in 'bla': undefined local variable or method 'x' for
main:Object (NameError)
```

Beim Vergleich von x mit $42$, versucht Ruby auf die (lokal) nicht gebundene Variable x zuzugreifen. Korrigieren, können wir das Programm, indem wir die Variable x als Parameter übergeben:

```
x = 42;

def bla(x)
  if x == 42
    then return(42);
    else return(0);
  end;
end;

puts(bla(x));
```

Hierbei muss man aber das x innerhalb der Funktion von dem x außerhalb unterscheiden. Wir könnten den Parameter auch anders bennenen, ohne dass sich das Programmverhalten ändern würde (Umbenennung lokaler Variablen hat keinen Effekt).

Dass Funktionen abgeschlossene Einheiten darstellen, erkennt man nicht nur daran, dass Variablen von außerhalb nicht im Funktionsrumpf bekannt sind. Umgekehrt sind auch Variablenbindungen, die innerhalb des Funktionsrumpfs vorgenommen wurden, nicht außerhalb der Funktion sichtbar:

```
x = 41;

def bla(x)
  x = x + 1;
  y = x;
  if x == 42
    then return(42);
    else return(0);
  end;
end;

puts(bla(x));   ⇝ 42
puts(x);        ⇝ 41
puts(y);        ⇝ Fehlermeldung
```

Dieses Programm gibt also zunächst 42 aus, dann 41 und bricht dann mit der Fehlermeldung

```
func.rb:14:in '<main>': undefined local variable or method 'y'
for main:Object (NameError)
```

ab.

### 2.8.2 Prozeduren

Im Gegensatz zu Funktionen haben Prozeduren keinen Rückgabewert, das bedeutet, dass sie auch keine Ergebnisse liefern (keine *return*!). Auch in Prozeduren können Variablen aus aufgerufenem Programmcode nicht gelesen oder verändert werden. Aber Prozeduren können z.B. Ausgaben erzeugen.

**Bsp.:** Ausgabe eines Quadrats gegebener Größe

```
def put_quadrat(n)
  line = "";              #
  for i in 1 .. n do      # line = "X" * n;
    line = line + "X";    #
  end;                    #
  for i in 1 .. n do
    puts(line);
  end;
end;

put_quadrat(5);

n = gets().to_i;
put_quadrat(n);
```

Da Prozeduren keinen Ergebniswert haben (sie machen höchstens Ausgaben auf dem Bildschirm, welche aber nicht als Ergebnis der Unterberechnung weiterverwendet werden können), können Prozeduren nur als Anweisungen verwendet werden:

```
n = gets.to_i;
put_quadrat(n);

for i in 0..5 do
  put_quadrat(i);
end;
```

### 2.8.3 Funktion mit Ausgabe

Eine Funktion, welche sowohl Ausgabe macht, als auch ein Ergebnis liefert, ist die Definition einer komfortableren Eingabe mit Erläuterungstext (Prompt):

**Bsp.:** Eingabe mit Prompt

```
def get_p(prompt)
  puts(prompt); # print(promt);
  return gets;
end;

n = get_p("Zahl:").to_i;
str = get_p("Name:").chop;
```

Beobachtung: Zeilenumbruch unschön ⇒ besser *print* anstelle von *puts* verwenden.

**Beachte:** Prozeduren und Funktionen können auch andere Prozeduren und Funktionen aufrufen.

**Bsp.:**

```
def n_spaces_in_between(str,n)
  return (str + "␣" * n + str);
end;

def triangle(n)
  puts("*");
  for i in 0 .. n-3 do
    puts(n_spaces_in_between("*",i));
  end;
  puts("*" * n);
end;

triangle(3);
puts();          #erzeuge eine Leerzeile/Zeilenumbruch
triangle(7);
```

Wenn Funktionen andere Funktionen aufrufen können, können sie sich dann auch selbst aufrufen?

Antwort: Ja, **Rekursion**

## 2.9 Rekursion

**Bsp.:** Fakultätsfunktion

Die Fakultätsfunktion ist ja mathematisch definiert als:

$$n! = \begin{cases} 1, \text{ falls } n = 0 \\ n \cdot (n-1)! \text{ , sonst} \end{cases}$$

Dies lässt sich direkt als rekursive Funktion umsetzen:

```
def fac(n)
  if n==0 then return 1;
          else return n * fac(n-1);
  end;
end;

puts(fac(5));    # 120
puts(fac(4));    # 24
```

Das Verhalten eines rekursiven Programms kann am Beispiel des Aufrufs puts(fac(3)); wie folgt verdeutlicht werden:

puts(fac(3)); hier muss zunächst das Ergebnis von fac(3) bestimmt werden:
fac(3)
↪ Code von fac mit Belegung n=3 auswerten
die Bedingung ist nicht erfüllt, deshalb weiter im else-Zweig
return 3*fac(2); hier muss zunächst das Ergebnis von fac(2) bestimmt werden:
fac(2)
↪ Code von fac mit Belegung n=2 auswerten
die Bedingung ist nicht erfüllt, deshalb weiter im else-Zweig
return 2*fac(1); hier muss zunächst das Ergebnis von fac(1) bestimmt werden:
fac(1)
↪ Code von fac mit Belegung n=1 auswerten
die Bedingung ist nicht erfüllt, deshalb weiter im else-Zweig
return 1*fac(0); hier muss zunächst das Ergebnis von fac(0) bestimmt werden:
fac(0)
↪ Code von fac mit Belegung n=0 auswerten
die Bedingung ist erfüllt, deshalb weiter im then-Zweig
return 1; die Rekursion terminiert, Ergebniswert bekannt
↩ der Aufruf von fac(0) wird durch 1 ersetzt
return 1*1; ausrechnen: 1*1 → 1
return 1; die Rekursion terminiert, Ergebniswert bekannt
↩ der Aufruf von fac(1) wird durch 1 ersetzt
return 2*1; ausrechnen: 2*1 → 2
return 2; die Rekursion terminiert, Ergebniswert bekannt
↩ der Aufruf von fac(2) wird durch 2 ersetzt
return 3*2; ausrechnen: 3*2 → 6
return 6; die Rekursion terminiert, Ergebniswert bekannt
↩ der Aufruf von fac(3) wird durch 6 ersetzt
puts(6); ausgeben ⇝ 6

Beachte, dass es nicht immer der Fall sein muss, dass eine Funktion nach dem rekursiven Aufruf selber auch direkt terminieren muss. Hier können noch weitere Funktionen (auch rekursiv!) aufgerufen werden.

↓↓↓↓↓ Zusatzinhalt, der für 5 ECTSler nicht relevant ist ↓↓↓↓↓

Zur Implementierung der Rekursion wird ein Stack (Keller), wie wir ihn in der Stack-Maschine kennen gelernt haben, verwendet. Bei der Ausführung von Prozeduren und Funktionen nennt man diesen Laufzeitstack (oder Laufzeitkeller). Auf ihm wird zu einem Funktionsaufruf die umgebenden Berechnungen gespeichert. Nach der Beendigung einer (nicht nur rekursiven) Funktion wird die Berechnung in der auf dem Laufzeitkeller gespeicherten Umgebung fortgesetzt. In der Ausführung oben haben wir dies durch die Einrückung dargestellt, d.h. weniger stark eingerückte Ausführungeteile liegen noch auf dem Laufzeitkeller und werden noch fertig ausgeführt, wenn die Unterberechnung (z.B. fac(2)) beendet wurde.

↑↑↑↑↑ Zusatzinhalt, der für 5 ECTSler nicht relevant ist ↑↑↑↑↑

**Bemerkung: Rekursion** ist genauso ausdrucksstark wie die Verwendung von Schleifen, welche auch als **Iteration** bezeichnet wird. Es gibt sogar Programmiersprachen, welche nur Rekursion und keine Schleifen anbieten (z.B. Funktionale Programmiersprachen).

Um zu verstehen, wie man zu iterativen Programmen äquivalente rekursive Programme entwickeln kann, betrachten wir noch einmal das Umdrehen eines Strings:

```
def reverse(str)
  if str.length<=1 then
    return str;
  else
    return reverse(str[1,str.length-1])+str[0,1];
  end;
end;

puts(reverse("abcde"));
```

Alternativ können wir auch den String unverändet lassen und im rekursiven Aufruf nur den Index, welchen wir im aktuellen Schritt übernehmen wollen, runterzählen:

```
def rev(str,i)
  if i<0 then
    return "";
  else
    return str[i,1]+rev(str,i-1);
  end;
end;

def reverse(str)
  return rev(str,str.length-1);
end;
```

Da neben dem Index auch der String, welchen wir umdrehen wollen, im Rumpf der rekursiven Funktion bekannt sein muss, erhält die Funktion rev zwei Parameter. Die eigentliche Funktion reverse mit einem Argument startet dann die zweistellige Hilfsfunktion rev mit einem passenden Wert für den ersten Index.

## 2.10 Objekte und ihre Identität

In Kapitel 3.3 hatten wir bereits Klassen, Objekte und Methoden kennengelernt. Insbesondere haben wir uns mit Methodenaufrufen vertraut gemacht, z.B. “$abc$“.$length \rightsquigarrow 3$

Objektorientierte Programmierung (Objektorientierte Programmiersprachen) macht aber mehr aus.

Objekte haben eine Identität und können verändert werden. Ihr Zustand ist durch ihre Attribute definiert.

**Bsp.:**
Bello könnte ein Objekt der Klasse Hund sein. Ein Attribut könnte z.B. sein Gewicht sein. Dann könnte es eine Methode fressen geben, welche natürlich sein Gewicht verändert.

Ein anderes Beispiel ist ein Konto. Wenn man von einem Konto Geld abhebt oder Geld einzahlt, erhält man kein neues Konto. Vielmehr verändert sich der Zustand des Kontos, welcher durch ein Attribut, den Kontostand repräsentiert wird.

Entsprechend können auch einige der Ruby Objekte verändert werden. Ein wichtiges Beispiel hierbei sind Strings.

```
str = "bla";
str[1,1]="xx";
puts(str);          ⇝ bxxa
```

Hierbei bedeutet, str[p,l]=str', dass in str der Teilstring an der Position p mit der Länge l durch den String str' ersetzt wird. Wir erhalten also vom Prinzip den String str[0,p]+str'+str[p+l,str.length]. Allerdings nicht wie bisher als neues Objekt. Vielmehr wird der alte String (im Beispiel "bla") verändert.

Dies wird an folgendem Ruby-Programm deutlich:

```
x = "bla";
y = x;
y[1,1] = "";
puts(y);          ⇝ “ba“
puts(x);          ⇝ “ba“
```

Die Variablen verweisen beide auf das gleiche String-Objekt, welches mittels $[1,1] =$ an der Position 1 geändert wird!

Im Gegensatz zu

```
x = "bla";
y = "bla";
y[1,1] = "";
puts(y);        ⇝ “ba“
puts(x);        ⇝ “bla“
```

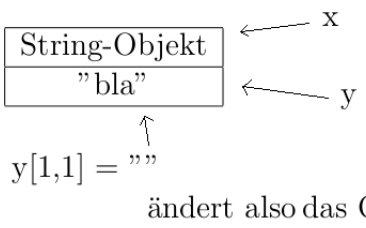
x und y zeigen auf zwei unterschiedliche Objekte, die zwar beide einen String "bla" darstellen, aber eben unterschiedliche Objekte sind (wie Zwillinge). Somit führt eine Veränderung des Objekts, auf das die Varibale y verweist, keiner Veränderung des anderen Objekts, auf das x verweist.

Entsprechend wollen wir zum Vergleich auch noch einmal das entsprechende Programm, welches die Stringkonkatenation verwendet, untersuchen.

```
x = "bla";
y = x;
y = y[0,1]+y[2,y.length];
puts(x);                    ⇝ "bla"
puts(y);                    ⇝ "ba"
```

Hier zeigen x und y zwar auf das gleiche String-Objekt. In der darauf folgenden Zuweisung y = y[0,1]+y[2,y.length]; wird aber kein Objekt verändert, sondern ein neues String-Objekt konstruiert und die Variable y verweist durch die Zuweisung auf dieses Objekt. x verweist weiter auf das alte, unveränderte String-Objekt.

Objekte besitzen also eine Identität und bestimmte Methoden (hier die 3-stellige Methode []=) können das Objekt verändern. Solche Methoden werden auch als *mutierende Methoden* (oder *destruktive Methoden*) bezeichnet. Variablen beinhalten nicht das gesamte Objekt sondern nur einen Verweis auf ein Objekt. Ähnlich wie bei der if−then−else-Funktion, wird dieser Methodenaufruf in Ruby in einer Mix-Fixnotation geschrieben, die syntaktisch zwar an die Zuweisung erinnern soll, aber in Wirklichkeit keine Zweisung ist.




Weitere Methoden der String-Objekte stehen ebenfalls in mutierenden Varianten zur Verfügung:

$$\begin{aligned} \text{“}abcdefg\text{”}.reverse &\rightsquigarrow \text{“}gfedcba\text{”} \\ \text{“}abcdefg\text{”}.reverse! &\rightsquigarrow \text{“}gfedcba\text{”} \end{aligned}$$

Die Methode $reverse$ liefert ein neues String-Objekt, während $reverse!$ das gegebene String-Objekt verändert. In Ruby existiert eine Konvention (Teil der Pragmatik von Ruby), dass der Name mutierender Methoden mit einem Ausrufezeichen endet. Hieran sollten sich alle Ruby-Programmierer halten! Der Unterschied der beiden Varianten, wird an folgenden Beispielprogrammen deutlich:

```
x = "abc";
y = x;
puts(y.reverse());      ⇝ cba
puts(x);                ⇝ abc
aber:
y.reverse!();           wie Prozedur
puts(y);                ⇝ cba
puts(x);                ⇝ cba
```

**Beachte:** Wenn man sich nicht für das Ergebnis eines Ausdrucks interessiert, kann man den Ausdruck in Ruby auch als Anweisung hinschreiben. Ergibt nur Sinn, wenn Effekte auf Objekten passieren oder Ausgaben, wie in Prozeduren.

```
3 + 4;  # <- unsinnig, da kein Objekt veraendert wird
x = 3;
```

aber

```
y.reverse!();    # <- sinnvoll, da das Objekt in y veraendert wird
puts(y);
```

Oft sind mutierende Methoden auch als Funktionen implementiert, die das veränderte Objekt auch als Ergebnis liefern. Da sie aber Objekte, welche außerhalb der Methodendefinition existieren, auch verändern, würde es eigentlich auch ausreichen, sie als Prozeduren ohne Rückgabewert zu definieren. Prozeduren können also nicht nur mehrere Ausgaben zusammenfassen. Sie können auch übergebene Objekte mutieren. Somit ergeben sich mit mutierenden Updates auch neue sinnvolle Möglichkeiten Prozeduren selber zu definieren.

Hier eine Übersicht über einige String-Methoden und ihre mutierenden Varianten:

- *capitalize* / *capitalize*!
- *chop* / *chop*!
- *delete* / *delete*! (löscht alle Character des Argumentstrings)
- *downcase* / *downcase*!
- *upcase* / *upcase*!
- *slice* (Methode zum Selektieren von Teilstrings) / *slice*! (Methode zum Löschen von Teilstrings)
- *strip* / *strip*!

Mit [] = können Teilstrings ersetzt werden. Zur besseren Verständlichkeit wird diese Methode meist Mixfix notiert:

```
str = "abcdef";
str[2,3] = "xx"; # str.[]=(2,3,"xx")
puts(str);
```

$\rightsquigarrow$ *abxxf*

Als Beispiel wollen wir nun eine Variante des *reverse* Programms implementieren, die den String verändert:

```
def rev!(str)
  for i in 0 .. str.length-1 do
    str[i,1] = str[str.length-i-1,1];
  end;
end; # return str
```

Das ! im Prozedurnamen soll zeigen, dass die definierte Prozedur ihr Argument verändert. Wir verwenden keinen Rückgabewert.

```
s = "1234";
rev!(s);
puts(s);          ~> 4334
s = "12345";
rev!(s);
puts(s);          ~> 54345
```

Das Programm verhält sich also noch nicht korrekt. Der Grund ist, dass das Kopieren zunächst Werte überschreibt, die später noch gebraucht werden.

Lösung: Tauschen von Werten


```
def rev!(str)
  for i in 0 .. str.length/2 - 1 do
    h = str[str.length-i-1,1];
    str[str.length-i-1,1] = str[i,1];
    str[i,1] = h;
  end;
end;

s = "1234";
rev!(s);
puts(s);                                    ⇝ 4321
```

Zum Tauschen zweier Werte verwendet man in der Regel eine Hilfsvariable (hier h), die den Wert, welcher zuerst überschrieben wird, zwischenspeichert.

Klassen, bei welchen alle Methoden, ein Objekt nicht mutieren, nennt mann auch *immutable*. Beispiele sind alle Klassen für Zahlen, wie wir sie bisher kennen gelernt haben und die Klassen für boolesche Werte. In einigen anderen Programmiersprachen (z.B. Java) sind auch Strings immutable. Aber eben nicht in Ruby.

Die Kombination von mutierenden und nicht mutierenden Methoden innerhalb einer Klasse birgt einige Gefahren, welche wir uns später noch genauer ansehen werden.

↓↓↓↓↓ Zusatzinhalt, der für 5 ECTSler nicht relevant ist ↓↓↓↓↓

## 2.11 Objektorientierte Datenmodellierung

Nachdem wir die Eigenheiten der Objektidentität und mutierenden Methoden verstanden haben, wollen wir uns die Definition eigener Klassen und Objekte anschauen.

In einer Klasse werden Objekte gleicher Art zusammen gefasst. Die Klasse stellt sozusagen das Gerüst für Objekte gleicher Bauart dar. Die Daten, welche in einem Objekt einer Klasse gespeichert werden, heißen Attribute und können in jedem einzelnen Objekt mit anderen Werten belegt werden. Eine Klassedefinition verfügt außerdem über Methoden, welche später für jedes konkrete Objekt der Klasse aufgerufen werden können und über die der Zugriff auf die Attribute erfolgen kann. Als erstes Beispiel wollen wir als neuen Zahlentyp Brüche (Fractional) definieren.

Zunächst definieren wir das Gerüst der Klasse Fractional. Als Konvention beginnen Klassennamen in Ruby mit einem Großbuchstaben.

```
class Fractional

  def initialize(zaehler,nenner)
    @zaehler = zaehler;
    @nenner  = nenner;
  end;

end;
```

Die spezielle Methode initialize heißt in der objektorientierten Programmierung *Konstruktor* und wird aufgerufen, wenn ein neues Objekt der Klasse Fractional angelegt wird. Zur Generierung eines Bruchs verwenden wir zwei Argumente, den Zähler und den Nenner. Beide übergebenen Werte speichern wir in Atributvariablen. Diese Attributvariablen werden durch ein @ am Anfang des Variablennamens gekennzeichnet und sind (nach ihrer Initialisierung) in der gesamten Klassendefinition gültig, also auch außerhalb des Konstruktors, indem sie iniitalisiert werden. Die Attribute werden also dafür verwendet, den eigentlichen Zustand der Objekte zu speichern.

Als nächstes wollen wir eine Methode zur Klassendefinition hinzufügen, welche einen Bruch ausgibt:

```
def show
  puts(@zaehler.to_s+"/"+@nenner.to_s);
end;
```

Dann können wir unsere Klasse bereits wie folgt verwenden:

```
x = Fractional.new(5,3);
x.show;                    # --> 5/3
```

Der Konstruktoraufruf Fractional.new erzeugt ein neues Objekt, welches dann durch den Konstruktor der Klasse Fractional initialisiert wird. Für eigene Objekte erkennt man so ganz genau, an welchen Stellen neue Objekte erstellt werden. Für vordefinierte Objekte, erfolgt dies implizit, wie z.B. bei 42 oder "Hallo".

In Ruby ist es aber eher ungewöhnlich, spezielle Ausgabemethoden zu definieren. Statt dessen definiert man für seine Objekte meist eine spezielle Methode zur Umwandlung in einen String, welcher dann ausgegeben werden kann. Hierbei können wir direkt auch unsere Ausgabe optimieren und unechte Brüche (Zähler > Nenner) als gemischte Brüche ausgeben.

```
def to_s
  if @nenner==1 then return @zaehler.to_s;
  else return (if @zaehler>@nenner then
                 (@zaehler/@nenner).to_s+" "
               else "" end +
               (@zaehler % @nenner).to_s+"/"+@nenner.to_s);
  end;
end;
```

Dann können wir unsere Klasse bereits wie folgt verwenden:

```
x = Fractional.new(5,3);
puts(x);                   # --> 1 1/3
```

Ein explizites to_s ist hier nicht notwendig, da die Prozedur puts ihr Argument automatisch mittels to_s in einen String umwandelt, falls es noch kein String ist.

Unschön ist aber noch, dass es mehrere redundante Darstellungen der gleichen Brüche gibt:

```
x = Fractional.new(5,3);
y = Fractional.new(10,6);
puts(x);                   # --> 1 1/3
puts(y);                   # --> 1 2/6
```

Es wäre besser, wenn Brüche immer direkt gekürzt würden, wenn sie angelegt werden. Dies ist aber mit Hilfe der ggt-Funktion kein großes Problem:

```
def initialize(zaehler,nenner)
  if nenner==0 then
    puts("Achtung Division durch 0!");
  end;
  ggt = ggt(zaehler,nenner);
  @zaehler = zaehler/ggt;
  @nenner = nenner/ggt;
end;

def ggt(a,b)
  while a!=0 && b!=0 do
    if a>b then a=a-b;
           else b=b-a;
    end;
  end;
  if a==0 then return b;
          else return a;
  end;
end;
```

Wir erhalten in obigem Programm nun auch für den Bruch Fractional.new(10,6) die Ausgabe 1 1/3.

Im nächsten Schritt können wir versuchen die Multiplikation von Brüchen zu definieren. Wie immer benötigt der Operator * einen weiteren Bruch als Argument. Da die Methode * das Objekt selber nicht mutieren soll, müssen wir hier ein neues Fractional Objekt anlegen, welches das Produkt der beiden Brüche repräsentiert:

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

Wie alle Zahlen, sollten auch unsere Brüche immutable sein.

Dies funktioniert so aber leider nicht, da die Attribute der Klasse Fractional nicht nach außen sichbar sind. Sie dürfen nur innerhalb der Klassendefinition verwendet werden. Wir benötigen also zwei zusätzliche Methoden, welche es uns ermöglichen auf diese beiden Attribute (lesend) zuzugreifen. Solche Methoden heißen *getter-Methoden* und stellen sich in unserem Beispiel wie folgt dar:

```
def nenner
  return @nenner;
end;

def zaehler
  return @zaehler;
end;
```

Dann funktioniert obige Definition der Multiplikation und wir erhalten:

```
x = Fractional.new(5,3);
y = Fractional.new(6,10);
puts((x*y));                    # --> 1
```

Beachte, dass es nicht notwendig ist, das Ergebnis noch einmal explizit zu kürzen, da der Konstruktor des neuen Objekts dies für uns übernimmt.

Entsprechend können wir auch die Addition definieren:

```
def +(o)
  return Fractional.new(@zaehler*o.nenner+o.zaehler*@nenner,
                        @nenner*o.nenner);
end;
```

Ähnlich, wie der Zugriff auf die Attribute normaler Weise verboten ist, wäre es natürlich auch schön, wenn wir bestimmte Methoden, wie den ggT, nur für den internen Gebracuh definieren könnten. Hierdurch ergäbe sich eine kleinere besser verständliche Schnitstelle der Klasse. Ruby bietet hierzu die Möglichkeit, bestimmte Methoden als private oder public zu deklarieren. Hierbei ist public der Default für alle Methoden. Durch das Schlüsselwort private werden alle folgenden Methodendefinitionen privat, bis entweder die Klassendefinition zu Ende ist oder das Schlüsselwort public kommt, wie wir an der finalen Version der Fractional-Klasse sehen:

```
class Fractional
  def initialize(zaehler,nenner)
    if nenner==0 then
      puts("Achtung Division durch 0!");
    end;
    ggt = ggt(zaehler,nenner);
    @zaehler = zaehler/ggt;
    @nenner = nenner/ggt;
  end;

  private

  def ggt(a,b)
    while a!=0 && b!=0 do
      if a>b then a=a-b;
             else b=b-a;
      end;
    end;
    if a==0 then return b;
            else return a;
    end;
  end;

  public

  def nenner
    return @nenner;
  end;
```

```
  def zaehler
    return @zaehler;
  end;

  def to_s
    if @nenner==1 then return @zaehler.to_s;
    else
      return (if @zaehler>@nenner then
                (@zaehler/@nenner).to_s+" "
              else ""
              end + (@zaehler % @nenner).to_s+"/"+@nenner.to_s);
    end;
  end;

  def +(o)
    return Fractional.new(@zaehler*o.nenner+o.zaehler*@nenner,
                          @nenner*o.nenner);
  end;

  def *(o)
    return Fractional.new(@zaehler*o.zaehler,@nenner*o.nenner);
  end;
end;

x = Fractional.new(2,3);
y = Fractional.new(5,6);

puts(x+y);
```

Als weiteres Beispiel wollen wir eine Klasse definieren, bei der es mutierende Methoden gibt: ein Konto.

```
class Konto

  def initialize(betrag)
    @kontostand = betrag;
  end;

  def get_kontostand
    return @kontostand;
  end;

  def abheben!(betrag)
    @kontostand = @kontostand - betrag;
  end;

  def einzahlen!(betrag)
    @kontostand = @kontostand + betrag;
  end;
end;
```

Um den aktuellen Kontostand sehen zu können, benötigen wir wieder eine getter-Methode. Die Methoden abheben! und einzahlen! verändern den Zustand unserer Objekte, d.h. es wird keine neues Objekt konstruiert. Vielmehr wird unsere Attributvariable @kontostand verändert. Deshalb benennen wir diese Methoden auch, wie in Ruby üblich, mit einem Ausrufezeichen.

```
k1 = Konto.new(200);
k2 = Konto.new(50);
k1.abheben!(50);
puts(k1.get_kontostand);                # --> 150
k2.einzahlen!(1000);
puts(k2.get_kontostand);                # --> 1050
```

Das Beispiel verdeutlicht auch die Objektidentität unserer Konten. Wir könenn mehrere Konten anlegen, welche alle einzelnen, unabhängig von einander verändert werden können.

↑↑↑↑↑ Zusatzinhalt, der für 5 ECTSler nicht relevant ist ↑↑↑↑↑

## 2.12 Mutierende und nicht mutierende Methoden

Wir haben mit mutierenden und nicht mutierenden Methoden zwei unterschiedliche Programmierstile kennen gelernt, welche in der objektorientierten Programmierung kombiniert werden. Es zeigt sich aber, dass die Kombination der beiden Ansätze neue Gefahren birgt, wo man sie bisher eigentlich nicht vermutet hat. Als Beispiel wollen wir ein Programm schreiben, welches zu einer eingegebenen Zeichenreihe ein Palindrom bildet. Hierzu definieren wir eine Hilfsfunktion rev, welche einen String umdreht (solch eine Funktion ist natrürlich schon vordefiniert, wir wollen Sie aber hier selber definieren um an diesem einfachen Beispiel das Problem zu analysieren). Danach können wir dann den String und seine umgedrehte Variante konkatenieren und erhalten das passende Palindrom.

```
def rev(str)
  for i in 1..str.length-1 do
    str = str[i,1]+str[0,i]+str[i+1,str.length]
  end
  return str
end

puts("Bitte gib eine Zeichenreihe ein:")
str = gets.chop
pal = rev(str)
pal[0,0] = str

puts("Das zu "+str+" passende Palindrom lautet: "+pal)
```

Bei der Funktion rev entschließen wir uns eine nicht mutierende-Variante zu programmieren. Diese sieht zwar anders aus, als unsere bisherigen Varianten, scheint aber durchaus korrekt zu sein, wie auch einige Testläufe (Vorlesung) zeigen. Die Idee ist, dass wir im Rumpf der Schleife einen neuen String zusammen stetzen, der aus dem Zeichen an der i-ten Position, gefolgt von den Zeichen vor dem i-ten Zeichen und den Zeichen hinter dem i-ten Zeichen zusammen gebaut wird.

Für das Zusammenfügen der beiden Strings in der Variablen pal verwenden wir eine mutierende Variante, welche den String str in den String pal vorne einfügt.

Führen wir das Programm aus, verhält es sich auch tatsächlich, wie wir es erwarten würden:

```
Bitte gib eine Zeichenreihe ein:
ot
Das zu ot passende Palindrom lautet: otto
```

und

```
Bitte gib eine Zeichenreihe ein:
ruby
Das zu ruby passende Palindrom lautet: rubyybur
```

Wenn wir aber eine Zeichenreihe der Länge eins eingeben geschieht das folgende:

```
Bitte gib eine Zeichenreihe ein:
a
Das zu aa passende Palindrom lautet: aa
```

Auch der Inhalt der Variablen str wurde durch die Mutation des Objekts in der Variablen pal verändert. Der Grund ist, dass die Funktion rev im Fall eines Strings der Längen null oder eins, einfach sein Argument (str) unverändert zurück gibt. Dies bedeutet dann natürlich, dass beide Variablen in diesen Fällen genau auf das gleiche Objekt verweisen. Wird dann eines der beiden Objekte verändert, ändert sich das andere Objekt mit.

Es zeigt sich also, dass wir auch bei der Programmierung von nicht mutierenden Funktionen und Methoden, sehr vorsichtig vorgehen müssen, wenn im System weitere mutierende Methoden für denselben Datentypen existieren. Untersucht man die anderen nicht mutierenden Methoden für Strings, so sieht man, dass Ruby hierbei tatsächlich vorsichtiger vorgeht, wie die folgenden Beispiele zeigen:

```
str = "abc"
str1 = str+""    # str = str*1 oder str = ""+str oder str.rev
str[1,1] = ""
puts(str)   #
puts(str1)  # bleibt immer unveraendert
```

Der String str1 bleibt also in allen Fällen von der Veränderung des Objekts in str in der dirtten Zeile unverändert. Alle verwendenten nicht mutierenden Methoden der Klasse String liefern eine Kopie ihres Parameterobjekts.

Für unser Programm bedeutet dies also, dass wir dafür sorgen müssen, dass unsere Funktion rev auch für die Fälle, in denen der übergebene String eine Länge kleiner gleich eins hat, sein Argument kopiert. Objektorientierte Programmiersprachen bieten hierzu meist Methoden zum *Klonen* von Objekten an: clone. Diese Operation können wir in der ersten Zeile unseres Programms verwenden, um die Definition von rev zu korrigieren:

```
def rev(str)
  str = str.clone
  for i in 1..str.length-1 do
    str = str[i,1]+str[0,i]+str[i+1,str.length]
  end
  return str
end
```

Dann erhalten wir auch für Eingaben der Länge eins ein korrektes Ergebnis

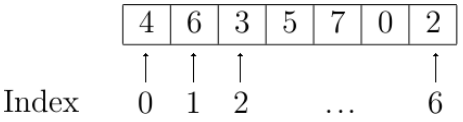
```
Bitte gib eine Zeichenreihe ein:
a
Das zu a passende Palindrom lautet: aa
```

Bei der Programmierung von nicht mutierenden Methoden muss man also folgendes beachten: Falls es für den gleichen Datentypen auch noch mutierende Methoden gibt, müssen die nicht mutierenden Methoden für alle Eingaben neue Objekte als Ergebnis liefern, d.h. sie dürfen keine übergebenen Objekte unkopiert zurückgeben. Funktionen und Methoden, die sich an diese Regel nicht halten, müssen als inkorrekt angesehen werden.

# 3 Datenstrukturen und Algorithmen

Bisher haben wir nur vordefinierte Datentypen kennengelernt. Oft ist es aber auch wichtig mehrere Werte zu neuen zusammengesetzten Werten zu machen. Ein gängiger Datentyp in imperativen Sprachen ist das Array, welches mehrere Werte in einem Speicherbereich hintereinander repräsentiert und die einzelnen Werte über einen Index adressierbar macht.

## 3.1 Arrays




Darstellung: $[4, 6, 3, 5, 7, 0, 2]$

Auf einen Index kann man mittels $a[i]$ zugreifen, wobei $a$ mit einem Array belegt und $i$ ein Index ist. Arrays können mittels $a[i] =$ auch verändert werden.

Ein neues Array kann mit folgenden Methoden konstruiert werden:

$$Array.new(n) \quad \rightsquigarrow \quad \underbrace{[nil, nil, ..., nil]}_{n-\text{mal}}$$

oder

$$Array.new(n, v) \quad \rightsquigarrow \quad \underbrace{[v, v, ..., v]}_{n-\text{mal}}$$

Außerdem ist es möglich, kleine Arrays konkret anzugeben, was aber natürlich nur bei kleinen Arrays praktikabel ist:

$$a = [1, 2, 3, 4, 5, 6];$$

Als erstes Programm wollen wir eine Funktion definieren, welche ein Array von Zahlen nimmt und die Summe der Zahlen berechnet:

```
def sum_array(a)
  sum = 0;
  for i in 0 .. a.size() - 1 do # Methode zum Bestimmen der Arraygroesse
    sum = sum + a[i];
  end;
  return sum;
end;

puts(sum_array([1,2,3,4,5])); # -> 15
```

Nun wollen wir eine Funktion definieren, die zwei Zahlen $n$ und $m$ nimmt und ein Array mit den Zahlen $n$, $n+1$, ..., $m$ zurückgibt:

```
def from_to(n,m)
  a = Array.new(m-n+1);
  for i in 0 .. m-n do
    a[i] = n;
    n = n+1;
  end;
  return a;
end;
```

Die Einträge im Array werden durch die Anweisung `a[i] = n` gesetzt. Hierbei handelt es sich nicht um eine Zuweisung. Vielmehr verwenden wir die mutierende Methode `[]=` zur Modifikation des Array-Objekts. Sie verhält sich sehr ähnlich, wie die entsprechende Methode bei Strings, allerdings wird nur ein Element verändert, so dass es keinen zweiten Parameter für die Länge des zu ersetzenden Teils gibt. Beachte aber, dass Arrays, genau wie Strings, Objekte sind. Veränderungen an den Arrays werden in der Regel durch die mutierende Methoden `[]=` vorgenommen.

Als Test für unser Programm wollen wir einmal ein generiertes Array ausgeben:

```
puts(from_to(3,7));
```

Wir erhalten die folgende Ausgabe:

```
3
4
5
6
7
```

Diese Ausgabe hätten wir so nicht erwartet. Der Grund für diese ungewöhnliche Ausgabe ist, dass *puts* in Ruby so definiert ist, dass alle Elemente eines Arrays einzeln, aber jeweils mit einem Zeilenumbruch, ausgegeben werden. Entsprechend gibt *print* alle Werte eines Arrays ohne Zeilenumbruch aus, was meistens auch nicht sehr nützlich ist:

```
print([3,4,5,6,7]); ⇝ 34567
```

Besser ist die Verwendung von p(..), das Arrays genau so ausgibt, wie sie auch in Ruby definiert werden können:

```
p([3,4,5,6,7]); ⇝ [3,4,5,6,7]
```

Als nächstes Beispiel wollen wir das Maximum aller Elemente eines Arrays von Zahlen bestimmen:

```
def max(a)
  max = a[0];
  for i in 1 .. a.size()-1 do
    if a[i]>max then max = a[i];
    end;
  end;
  return max;
```

```
end;

puts(max([3,4,7,5]); # -> 7
```

Vergleicht man Arrays mit Strings, so fallen gewissen Ähnlichkeiten auf. Strings entsprechen Arrays von Zeichen und so ist es auch bei Strings möglich, einzelne Zeichen zu selektieren. Hierbei ist aber darauf zu achten, dass das Ergebnis der Selektion einzelner Zeichen je nach Ruby-Version entweder einen Character (genauer Strings der Länge eins) oder den ASCII-Wert des Characters liefert. Umgekehrt können in Ruby auch bei Arrays Teilarrays selektiert werden oder zwei Arrays mittels + konkateniert werden:

```
p([1,2,3]+[4,5]); ⇝ [1,2,3,4,5]
puts("Hallo"[1]); ⇝ "a" # bzw. 97 in Ruby 1.8
p([1,2,3,4,5][2,3]);  ⇝ [3,4,5]
```

Außerdem können auch Teilarrays modifiziert werden:

```
a = [1,2,3,4,5];
a[2,1] = [42,43,44];
p(a);  ⇝ [1,2,42,43,44,4,5]
```

Funktionen zum Selektieren, Zusammenfügen oder Ersetzen von Teilarrays sind nicht in allen Programmiersprachen verfügbar. In der Regel verwendet man nur Operationen, zum Nachschlagen oder Ersetzen einzelner Einträge des Arrays, da diese sehr effizient implementiert werden können. Hierzu später noch mehr.

Bisher haben wir Arrays in der Regel dazu verwendet, gleichartige Daten zusammenzufassen. Dies ist in Ruby aber nicht notwendig. Wir können auch unterschiedliche Arten von Daten in einem Array zusammenfassen. Dann sollten wir uns aber darüber im Klaren sein, an welcher Stelle wir welche Art von Daten abgelegt haben, um diese später sinnvoll verwenden zu können. Als Beispiel können wir ein Array verwenden um z.B. ein Tripel bestehend aus einem Namen (String), der Größe der Person (Zahl) und ihrer aktuellen Anwesenheit (booleschen Wert) zu definieren:

```
person1 = ["Frank",188,true];
person2 = ["Sandra",162,false];

if person1[1]>person2[1]
  then puts(person1[0]+" ist groesser als "+person2[0]);
  else puts(person2[0]+" ist groesser als "+person1[0]);
end;
```

↓↓↓↓↓ Zusatzinhalt, der für 5 ECTSler nicht relevant ist ↓↓↓↓↓

Eine alternative und sicherlich schönere Variante wäre natürlich die Definition einer eigenen Klasse *Person*, mit entsprechenden Attributen, getter/setter-Methoden.

↑↑↑↑↑ Zusatzinhalt, der für 5 ECTSler nicht relevant ist ↑↑↑↑↑

Da Arrays selber auch wieder Werte darstellen, können wir tatsächlich auch wieder Arrays als Einträge in einem Array verwenden, wie folgendes Beispiel verdeutlicht:

```
a = [2,true,[42,[3,"Hallo"],"Leute"],5.0];

puts(a.size);      # -> 4
```

```
puts(a[0]);          # -> 2
puts(a[1]);          # -> true
p(a[2]);             # -> [42,[3,"Hallo"],"Leute"]
puts(a[3]);          # -> 5.0

puts(a[2][2]);     # -> "Leute"
puts(a[2][1][1]); # -> "Hallo"
```

Wir können also durch mehrfache Array-Zugriffe auch nach und nach in der verschachtelten Array-Struktur absteigen.

Beachte folgenden semantischen Unterschied beim Einfügen eines Arrays in eine Array:

```
a = [1,2,3,4,5];
a[3,1] = [42,43,44]; # Ersetzen eines Teilarrays
a[2] = [0,0];          # Ersetzen eines Elements durch ein Array
p(a);  ~> [1,2,[0,0],3,42,43,44,5]
```

Betrachten wir noch einmal das vorherige Beispiel der Personen. Wenn wir nicht nur zwei Personen speichern wollen, sondern beliebig viele, sollten wir anstelle zweier konkreter Personen besser ein Array von Personen verwenden:

```
personen = [["Frank",188,true],["Sandra",162,false],
            ["Sandra",168,false]];
```

Wie kann man solch ein Array von Arrays interpretieren?

Es handelt sich um eine Tabelle, d.h. wir haben mehrere Zeilen, für die in jeder Spalte gleichartige Information steht, was man auch mit folgender Darstellung noch unterstreichen kann:

```
personen = [["Frank"  ,188,true ],
            ["Sandra",162,false],
            ["Sandro",168,false]];
```

Solche Tabellen kennen wir auch schon aus Tabellenkalkulationen. Haben alle Einträge im Array die gleiche Struktur nennt man solche Arrays auch mehrdimensionale Arrays (hier speziell zweidimensional). Einige Programmiersprachen unterstützen mehrdimensionale Arrays explizit und sichern durch statische Typisierung auch zu, dass alle inneren Arrays die gleiche Struktur und Größe besitzen. In Ruby ist dies nicht der Fall und der Programmierer ist hierfür selber verantwortlich.

Ein Austausch solcher Tabellen auch über verschiedene Programme hinweg, wäre natürlich wünschenswert. Hierzu verwendet man gerne Dateien des Formats *comma separated values* (csv), in welchem Tabellen zeilenweise in einer Datei gespeichert werden. Die einzelnen Spalten sind dann durch ein Trennsymbol, in der Regel ein Komma (in der Deutschen version von Tabellenkalkulationsprogrammen standardmäßig durch ein Semikolon) getrennt. In obigem Beispiel, sähe der zugehörige Dateiinhalt wie folgt aus:

```
Frank,188,true
Sandra,162,false
Sandro,168,false
```

Wie können wir nun solche Dateien in ein zweidimensionales Ruby-Array einlesen? Hierzu ist es hilfreich die vordefinierte String-Methode `split` zu verwenden. Die Semantik der Methode wird schnell an folgenden Beispielaufrufen klar:

```
"a,b,cd,e".split(",") ⇝ ["a", "b", "cd", "e"]
"abcdefcdgh".split("cd") ⇝ ["ab", "ef", "gh"]
"ab\ncd\nef\n".split("\n") ⇝ ["ab", "cd", "ef"]
```

Man beachte, dass im letzten Beispiel der leere String hinter dem Zeilenumbruch nicht als separater Eintrag ins Array eingefügt wird.

Mit Hilfe der split-Methode können wir nun sehr einfach einen gegebenen String (der den Inhalt einer csv-Datei repräsentiert) zunächst in seine Zeilen und dann jede Zeile in die einzelnen Spalten zerlegen und so ein zweidimensionales Array konstruieren. Wir stellen die entsprechende Funktion zum Einlesen direkt für eine csv-Datei zur Verfügung:

```
def csv_read(filename)
  str = File.read(filename);
  lines = str.split("\n");
  for i in 0..lines.size-1
    lines[i] = lines[i].split(",");
  end;
  return (lines);
end;

persons = csv_read("test.csv");

min_pos = 0;
min = persons[0][1];

for i in 1..persons.size-1 do
  if persons[i][1]<min
    then min = persons[i][1];
         min_pos = i;
  end;
end;

puts(persons[min_pos][0]+" ist am kuerzesten.");
```

Das Hauptprogram berechnet abschließend noch, wer die kürzeste Person aus der test-Datei ist.

Um in Ruby berechnete Tabellen auch in einer anderen Anwendung nutzen zu können, müssen diese auch als Ruby-Programmen abgespeichert werden können. Eine entsprechende Funktion csv_write können wir ähnlich einfach definieren. Hierzu benötigen wir noch eine Funktion zum Schreiben von Dateien. In Ruby bietet die Klasse File die zweistellige Funktion write zum Schreiben eines Strings in eine Datei an.

```
def csv_write(file,a)
  str = ""
  for i in 0..a.size-1
    for j in 0..a[i].size-2
      str = str + a[i][j] + ",";
    end;
    str = str + a[i][a[i].size-1] + "\n";
  end;
  File.write(file,str)
```

```
end;
```

Es gibt auch varianten von csv, bei denen als Separator der Spalten ein Semikolon anstelle eines Kommas verwendet wird (insb. in deutschsprachigen Anwendungen). Außerdem werden die Strings zwischen den Kommas oft auch in Anführungszeichen gesetzt, um in diesen z.B. auch Kommas zu erlauben. Dies berücksichtigt unsere hier vorgenommene csv-Implementierung noch nicht. Vordefinierte Implementierungen sind hier aber flexibler und können in solchen Fällen ähnlich einfach, wie die hier vorgestellten Funktionen benutzt werden.

In der Vorlesung entwickeln wir außerdem ein etwas komplexeres Beispiel, mit welchem Veranstaltungen im Universitätsbetrieb verwaltet werden können. Hierbei verwenden wir Tabellen (zweidimensionale Arrays), welche wir als csv-Datei auf der Festplatte speichern und so auch in einer Tabellenkalkulation bearbeiten können.

↓↓↓↓↓ Zusatzinhalt, der für 5 ECTSler nicht relevant ist ↓↓↓↓↓

## 3.2 Reguläre Ausdrücke

Ähnlich wie die EBNF sind reguläre Ausdrücke ein mächtiges Mittel zur Sprachbeschreibung, wenngleich sie doch weniger ausdrucksstark sind. Mit Hilfe von regulären Ausdrücken können, sehr einfach Teilausdrücke, aber auch bestimmte Muster von Zeichenfolgen in Zeichenreihen gefunden werden. Ein regulärer Ausdruck wird in Ruby durch /-Symbole begrenzt. Reguläre Ausdrücke sind auch Werte, d.h. sie können auch in Variablen und Datenstrukturen gespeichert werden.

Die einfachsten reguläre Ausdrücke sind Zeichenfolgen, z.B. /cd/ oder /Hallo/. Mit Hilfe der Methode match können reguläre Ausdrücke gegen Strings *gematcht* werden. Dies bezeichnet man auch als *Pattern Matching*, da der reguläre Ausdruck als Muster (engl. Pattern) in einem String gesucht wird (engl. match). Das Ergebnis des Matchings ist ein MatchData-Objekt, welches Informationen zum erfolgreichen Match enthält. Falls der reguläre Ausdruck an keiner Position des Srings matcht, liefert der Aufruf der match-Methode das Ergebnis nil:

```
m1 = /cd/.match("abcdcd")\\
puts(m1)   # -> <MatchData "cd">
m2 = /ce/.match("abcdef")\\
puts(m2)   # -> nil
```

Das MatchData-Objekt enthält eine Reihe von Informationen zum gematchten String:

- to_s gibt an, welcher String gematcht wurde
- pre_match gibt den String vor dem gematchten String an
- post_match gibt den String hinter dem gematchten String an

Über die Länge von pre_match erhalten wir auch die Position, an welcher gematcht wurde.

Diese Funktionen erscheint im Moment noch überflüssig, da wir bisher nur Teilstrings gesucht haben. Es wird aber nützlich sein, wenn allgemeinere Muster formuliert werden können, welche ggf. auch unterschiedliche Teilstrings matchen können. Als erstes allgemeineres Beispiel betrachten wir den regulären Ausdruck, der auf alle Ziffern passt $\backslash d$:

```
/c\dd/.match("abc7def").to_s   ⇝ "c7d"
```

In diesem Beispiel ist schon der gemathcte String interessant, da er z.B. aus einer Eingabe kommen kann.

Für andere Klassen von Zeichen gibt es entsprechende reguläre Ausdrücke, z.B. für Whitespaces:

```
/c\sd/.match("abc def").to_s    ⇝ "c d"
/c\sd/.match("abc\ndef").to_s   ⇝ "c d"
/c\sd/.match("abc \ndef")       ⇝ nil
```

Mit $\backslash w$ können Buchstaben und Ziffern gematcht werden:

```
/c\we/.match("abcdef").to_s    ⇝ "cde"
/c\wd/.match("abc3def").to_s   ⇝ "c3d"
/c\sd/.match("abc(de)f")       ⇝ nil
```

Der Punkt . steht für ein beliebiges Zeichen:

```
/c.e/.match("abcdef").to_s     ⇝ "cde"
```

Außerdem ist es möglich für einzelne Buchstabe Bereiche anzugeben, z.B. $[p-t]$ oder $[a-zA-Z]$ oder $[0-9]$ oder konkrete mögliche Buchstaben aufzuzählen, z.B. $[ruby]$:

```
/[c-d]/.match("abcdef").to_s     ⇝ "c"
/[c-d][c-d]/.match("abcdef").to_s    ⇝ "cd"
/[ruby]/.match("abcdef").to_s    ⇝ "b"
```

Außerdem können bestimmte Buchstaben oder Bereiche ausgenommen werden:

```
/[^ab]/.match("abcdef").to_s     ⇝ "c"
```

Es ist auch möglich, Alternativen zu definieren:

```
/c|d/.match("abcdef").to_s     ⇝ "c"
/c|d/.match("abCdef").to_s     ⇝ "d"
/Hi|Hallo/.match("Hi Willi").to_s     ⇝ "Hi"
/Hi|Hallo/.match("Hallo Willi").to_s     ⇝ "Hallo"
```

Der Alternativen-Operator | bindet hierbei schwächer als die Hintereinanderschreibung von Buchstaben, so dass

$/Hi|Hallo/$ identisch mit $/(Hi)|(Hallo)/$ ist, aber eben nicht identisch mit $/H(i|H)allo/$.

Mit Hilfe von Klammern können reguläre Ausdrücke auch strukturiert werden, z.B.:

```
/b(c|x)d/.match("abcdef").to_s     ⇝ "bcd"
/b(c|x)d/.match("abxdef").to_s     ⇝ "bxd"
```

Was bedeutet dieser reguläre Ausdruck: $/a(|x)b/$?

```
/a(|x)b/.match("ab").to_s     ⇝ "ab"
/a(|x)b/.match("axb").to_s    ⇝ "axb"
```

Links vom | steht hier der leere reguläre Ausdruck (entspricht dem leeren Wort), welcher auf den String der Länge Null match.

So kann man also einfach ausdrücken, dass etwas (beschrieben durch einen regulären Ausdruck) optional ist. Alternativ kann man hierfür auch das ? als nachgestellten Operator (Postfix) verwenden:

```
/cd?e/.match("abcdef").to_s      ⇝ "cde"
/cx?d/.match("abcdef") .to_s     ⇝ "cd"
/(cd)?e/.match("abcdef").to_s    ⇝ "cde"
/(cx)?e/.match("abcdef") .to_s   ⇝ "e"
```

Beachte, dass ? stärker bindet, als das Hintereinanderschreiben, sich also nur auf den letzten regulären Ausdruck vor dem ? bezieht (oft Buchstabe):

/ab?c/ ist also idetisch mit /a(b?)c/, aber eben nicht identisch mit /(ab)?c/.

Abschließend kann man noch wie bei der EBNF Wiederholungen $(+)$ oder optionale Wiederholungen $(*)$ zulassen. Man beachte, dass ein Stern in Postfixnotation anstelle geschweifter Klammern wie in der EBNF verwendet wird:

```
/cx*d/.match("abcdef").to_s      ⇝ "cd"
/cx+d/.match("abcdef")           ⇝ nil
/cx+d/.match("abcxxxdef").to_s   ⇝ "cxxxd"
```

Zusätzlich zum Überprüfen, ob ein regulärer Ausdruck passt und an welcher Stelle er passt möchte man häufig auch noch weitere informationen erhalten.

**Bsp.:**
Gegeben ist ein String str der Form

"Name: Telefonnummer\n Name: Telefonnummer\n"
(z.B.: "Frank: 7277\n Christoph Daniel: 7297\n")

Nun suchen wir Christoph Daniels Telefonnummer.

```
eintrag = /Christoph Daniel: [0-9]+/.match(str).to_s
                                  ⇝ "Christoph Daniel: 7297"
```

Hieraus können wir nun die Telefonummer extrahieren, z.B. wieder mit einen regulären Ausdruck:

```
/[0-9]+/.match(eintrag).to_s   ⇝ "7297"
```

Eine Alternative und pft noch komfortablere Möglich, dieses Problem zu lösen bilden sogenannte *capture groups*. Die Idee ist, dass bei Matchen des regulären Ausdrucks, auch die Information vorgehalten wird, gegen welche Teilstrings die verwendeten geklammerten Telausdrücke gematcht wurden. So können wir im Beispiel einfach die Telefonnummer im regulären Ausdruck klammern:

```
eintrag = /Christoph Daniel: ([0-9]+)/.match(str)
```

Im nächsten Schritt können wir dann die Telefonnummer einfach an einem Index abfragen:

```
telefonnummer = eintrag[1]
```

Die Idee ist, dass alle capture groups von links nach rechts durchnummeriert werden und das Matchobjekt diese an den entsprechenden Indizes zur Verfügung stellt. In der Realisierung leifert ein MatchData-Objekt eine Array, welches dann indiziert wird. Arrays werden wir

im nächsten Kapitel genauer kenen lernen und dann sicherlich auch die Notation besser verstehen können.

Das folgende Beispiel mit mehreren capture groups verdeutlicht die Indizes, über welche anschließend auf die Teilmatches zugegriffen werden kann:

**Bsp.:**

```
 1      23          4
 ↓      ↓↓          ↓
/(a|b)((b|a)+)(..)/.match("abbbacbb")[i]
```

liefert für die folgenden Position i: 1 belegt mit "a"
2 belegt mit "bbba
3 belegt mit "a" //Der letzte String, der $(b|a)$ gematcht hat
4 belegt mit "cb"

Als etwas komplexeres Beispiel wollen wir in einem Text ein Kennzeichen suchen. Kennzeichehn können wir mit dem folgenden regulären Ausdruck definieren.

```
regexp_kennzeichen = /[A–Z][A–Z]?[A–Z]?–[A–Z][A–Z]? [1–9][0–9]*/
```

Dann könenn wir z.B. einen Text aus einer Datei einlesen und alle vorkommenen Auto-Kennzeichen ausgeben:

```
str = File.read("datei_mit_kennzeichen.txt")

m = regexp_kennzeichen.match(str)

while m!=nil do
  puts(m)  # to_s koennen wir uns hier sparen,
           # da dies durch puts gemacht wird
  str = m.post_match
  m = regexp_kennzeichen.match(str)
end
```

Mit der String-Methode *sub* ist es außerdem möglich, das erste (maximale) Matching eines regulären Ausdrucks durch einen String zu ersetzen:

**Bsp.:**

```
"abcbcabcbcbc".sub(/(bc)+/, "x") ⇝ "axabcbcbc"
```

Mit *gsub* können entsprechend alle passenden Substrings ersetzt werden:

```
"abcbcabcbcbc".gsub(/(bc)+/, "x") ⇝ "axax"
```

Beachte, dass hierbei das Ergebnis nicht wieder gematcht wird:

```
"abcbcabcbcbc".gsub(/(bc)+/, "bc") ⇝ "abcabc"
```

*sub* und *gsub* erlauben auch Strings anstelle eines regulären Ausdrucks.

Reguläre Ausdrücke werden z.B. auch in Textverarbeitungssystemen eingesetzt. In der Vorlesung schauen wir uns hierzu den Suchen und Ersetzen Mechanismus von Komodo-edit genauer an.

↑↑↑↑↑ Zusatzinhalt, der für 5 ECTSler nicht relevant ist ↑↑↑↑↑

## 3.3 Datenbanken

Bisher haben wir die Strukturierung von Daten innerhalb einer Programmiersprache kennen gelernt. Unsere letzte Anwendung hat einen Eindruck gegeben, wie man dies mit Hilfe von Tabellen (csv) machen kann. Für komplexere Anwendungen reichen einzelne Tabellen aber nicht aus. Datenbanken bieten einem die Möglichkeit mehrere Tabellen zu kombinieren und effizient auf die gespeicherten Daten zuzugreifen. Ausgehend von unserer Chat-Anwendung wollen wir uns das Thema Datenbanken erarbeiten. Hierbei wollen wir nun die (vereinfachte) Datenbank einer Chat-Anwednung, wie sie z.B. auf einem Chat-Server im Internet vorgehalten wird modellieren.

In der Chat-Anwendung mit csv-Dateien, haben wir die Empfänger bzw. Sender einer Nachricht durch einen String identifiziert. In modernen Chat-Systemen sind hier bei jedem möglichen Chat-Partner aber weitere Informationen, wie sein Name, eine Telefonnummer und ggf. ein Bild und Status hinterlegt. Diese Werte möchten wir natürlich nicht bei jeder einzelnen NAchricht wieder mit abspeichern. Außerdem wird es dann schwierig, diese Daten zu ändern. Man müsste alle Nachrichten der entsprechenden Person ändern. Deshalb legt man für die Personen des Chat-Systems eine separate Tabelle an, in welcher man die speziellen Attribute der Personen Speichert und dann entsprechend nur hier ändern muss.

Bei der Angabe eines senders bzw. Empfängers einer Nachricht, können wir dann einfach auf die entsprechenden Personen verweisen. Hierzu müssen diese natürlich eindeutig identifiziert werden könnne, wozu man in der Datenbank einen speziellen Identifier (ID) verwendet. Bei gängigen Smartphone-Apps ist dieser Identifier die Handynummer. In gängigen Datenbank Systemen verwendet man spezielle Identifier, meist Integer-Werte, welche den Datensatz eindeutig identifizieren. Diese Personen-Identifier kann man dann als sender- bzw. empfaenger-Attribut bei den Nachrichten eintragen. Unsere Datensätze sehen dann wie folgt aus:

| Nachrichten |
|---|
| ID |
| zeit |
| text |
| sender |
| empfaenger |

| Personen |
|---|
| ID |
| name |
| ... |

Konkrete Tabellen könnten dann wie folgt aussehen:

Nachrichten:

| ID | zeit | text | sender | empfaenger |
|---|---|---|---|---|
| 0 | 16:16 | Hallo Willi | 0 | 2 |
| 1 | 17:20 | Selber hallo | 2 | 0 |
| 2 | 18:05 | Wie geht es? | 1 | 2 |
| 3 | 18:07 | Gut. Und selbst? | 2 | 1 |

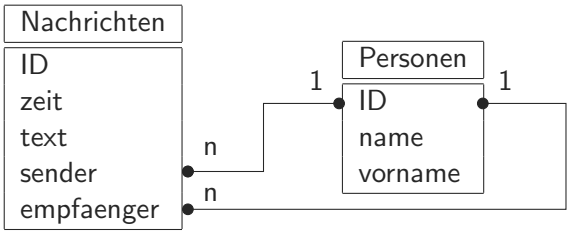
Personen:

| ID | name | vorname |
|---|---|---|
| 0 | Huch | Frank |
| 1 | Dylus | Sandra |
| 3 | Esquivel | Sandro |

Die IDs bei den Nachrichten sind im Moment noch nicht notwendig, schaden aber auch nicht.

Beachte, dass wir die IDs nur eindeutig für die jeweilige Tabelle verwenden. Dies reicht aus, da wir eine eindeutige Zuordnung zwischen der sender- bzw. empfaenger-Spalte in der Nachrichtentabelle und der Personen-ID haben. Es ist klar, zu welcher Tabelle diese ID gehört.

Im allgemeinen besteht eine Datenbank, ähnlich wie eine Tabellenkalkulation, aus einer Reihe von Tabellen, welche über IDs zusätzlich Verweise auf anderen Tabelleneinträge enthalten. Zur Modellierung solcher Datenbanken vewendet man gerne Datenbank-Schemata, in welchen Tabellen und Beziehungen zwischen den Tabellen graphisch ansprechend dargestellt werden können. Für unsere bisherige Anwendung ergibt sich das folgende Datenbank-Schema:




Jede Zeile entspricht einem Eintrag in die Tabelle. Die Reihenfolge ist eigentlich irrelevant, wichtig ist nur, dass wir die Zeilen unterscheiden können[1]. Hierfür eignen sich Namen in der Regel nicht, da es durchaus Menschen mit gleichen Namen gibt. Als Unteschеidungsmerkmal führt man deshalb in der Regel *Schlüssel* ein, welche es uns ermöglichen unserer Datensätze in jedem Fall zu unterscheiden. Solche Schlüssel kennen wir auch aus anderen Bereichen, wie z.B. die Matrikelnummer oder unsere Personalausweisnummer. Hier haben wir der Einfachheithalber eine Zahl hinzugefügt, welche nicht doppelt verwendet werden darf. Dass einzelne Zahlen fehlen stellt kein Problem dar und kann z.B. dadurch entstanden sein, dass eine Person, welche zuvor mit dem Schlüssel 2 existierte gelöscht wurde. Solche IDs werden von Datenbanksystemen in der Regel automatisch generiert. In unserer csv-Anwendung, mussten wir uns hierum noch selber kümmern.

Die Schlüssel benötigen wir außerdem, wie bereits oben motiviert, wenn wir Daten aus unseren beiden Tabellen miteinander verknüpfen. Untersucht man die Beziehung zwischen Nachrichten und Personen, so wird immer von genau einer Person gesendet und von genau einer Person empfangen. Eine Person kann aber durchaus mehrere Nachrichten denden und empfangen. Es gibt also zwei so genannte 1:n-Relation zwischen Nachrichten und Personen, welche man aus Sicht der Nachricht wirdgesendetVon/wirdempfangenVon und aus Sicht der Personen als sendet/empfaengt bezeichnen kann. Diese Relation können wir im Datenbank-Schema einfach dadurch ausdrücken, dass wir der Nachricht entsprechende Attribute hinzufügen, welche jeweils auf genau eine Person verweist. Eine Person kann so auch bei mehreren Nachrichten eingetragen werden, was genau die 1:n-Beziehung dieser Relation wiederspiegelt. Im Datenbank-Schema notieren wir die 1:n-Beziehung mit Hilfe der eingefügten Kanten.

In unserer csv-Implementierung haben wir das so implementiert, dass wir z.B. der send-Tabelle eine Spalte mit dem Empfämger einer Nachricht hinzugefügt haben. Möchte man diese Relation in umgekehrter Reihenfolge lesen (als alle Nachrichten, die an eine bestimmte Person geschickt wurden), kann man dies erreichen, in dem man in der send-Tabelle nach allen Nachrichten sucht, die den entsprechenden Empfänger haben.

Im folgenden werden wir ein einfaches Datenbanksystem kennenlernen. Wir verwenden SQ-

[1] In unserer bishrigen Implementierung mit csv-Dateien, spielt die Reihenfolge natürlich sehr wohl eine Rolle. Beim Übergang zu Datenbanken wird sie aber unwichtig werden.

Lite (genauer sqlite3) und realisieren als Beispiel dieses Datenbankschema. Gesteuert werden Datenbanken in der Regel durch *Structured Query Language (SQL)*. Die wichtigsten Konstrukte wollen wir schrittweise in SQLite kennen lernen.

Zunächst starten wir SQLite, wobei wir als Parameter den Namen der Datenbank angeben, mit der wir arbeiten wollen, mittels `sqlite3 chat`. Danach können wir interaktiv mit der Datenbank arbeiten und zunächst eine Tabelle für Personen anlegen:

```
sqlite> CREATE TABLE personen (id INTEGER PRIMARY KEY,
                               name TEXT, vorname TEXT);
```

Die Schlüsselworte schreibt man in SQL in der Regel mit Großbuchstaben. SQLite erlaubt hier aber auch Kleinbuchstaben.

Mit diesem ersten SQL-Befehl generieren wir eine Tabelle mit dem Namen personen, welche drei Spalten (id, name und vorname) hat. Als id verwedet man in der Regel Zahlen, was wir mit dem Typ INTEGER für die erste Spalte festlegen. Außerdem legen wir fest, dass die id ein Primärschlüssel ist und vom Datenbanksystem automatisch incrementiert wird. In die beiden weiteren Spalten können Strings (Typ TEXT) geschrieben werden.

Nun können wir einen ersten Datensatz in diese Tabelle eintragen:

```
sqlite> INSERT INTO personen (id, name, vorname) VALUES
        (1,"Huch", "Frank");
```

Mit Hilfe der Spaltenbeschriftung zwischen dem Tabellennamen und dem Schlüsselwort VALUES, können wir die Reihenfolge der Werte bestimmen, welche wir eintragen wollen. Außerdem können wir einzelne Spalten leer lassen, was dazu führt, dass entweder Default-Werte oder NULL-Werte eingetragen werden. Werden alle Werte in der Reihenfolge, wie sie beim Anlegen der Tabelle definiert wurden, eingetragen, kann des Tupel mit den Spaltennamen auch weggelassen werden:

```
sqlite> INSERT INTO personen VALUES (2,"Dylus", "Sandra");
sqlite> INSERT INTO personen VALUES (3,"Esquivel", "Sandro");
```

Versucht man einen Wert mit einem bereits vergebenen Schlüssel einzutragen, führt dies auf Grund der Primärschlüssel-Eigenschaft des id-Attributes zu einer Fehlermeldung:

```
sqlite> INSERT INTO personen (id, name, vorname) VALUES
        (2,"Merkel", "Angela");
Error: UNIQUE constraint failed: personen.id
```

Auf Grund der Angabe PRIMARY KEY bei der Definition der id-Spalte, haben wir aber auch die Möglichkeit, das System eine geeignete id generieren zu lassen, wenn wir keinen passenden Wert angeben:

```
sqlite> INSERT INTO personen (name, vorname) VALUES
        ("Merkel", "Angela");
```

Nun können wir unsere erste Anfrage an die Datenbank stellen und alle Spalten der Tabelle personen ausgeben:

```
sqlite> SELECT * FROM personen;
1|Huch|Frank
2|Dylus|Sandra
3|Esquivel|Sandro
4|Merkel|Angela
```

Die Verwendung des * bedeutet hierbei, dass alle Spalten in de Reihenfolge, wie sie bei der Generierung der Tabelle angegeben wurden, ausgegeben werden. Wir können uns aber auch auf die Ausgabe einzelner Spalten einschränken und auch deren Reihenfolge verändern:

```
sqlite> SELECT vorname,name FROM personen;
Huch|Frank
Dylus|Sandra
Esquivel|Sandro
Merkel|Angela
```

Nun können wir eine zweite Tabelle für die Nachrichten anlegen:

```
sqlite> CREATE TABLE nachrichten
            (id INTEGER PRIMARY KEY AUTOINCREMENT,
             zeit DATETIME, text TEXT, sender_id INTEGER,
             empfaenger_id INTEGER);
sqlite> INSERT INTO nachrichten (zeit, text, sender_id, empfaenger_id)
        VALUES ("2016-01-01 16:16:16","Hallo Willi",2 , 1);
sqlite> INSERT INTO nachrichten (zeit, text, sender_id, empfaenger_id)
        VALUES (datetime('now'),"Selber hallo",1,2);
sqlite> INSERT INTO nachrichten (zeit, text, sender_id, empfaenger_id)
        VALUES (datetime('now'),"Wie geht es?", 3, 2);
sqlite> INSERT INTO nachrichten (zeit, text, sender_id, empfaenger_id)
        VALUES (datetime('now'),"Gut und selber", 2, 3);
sqlite> SELECT * FROM nachrichten;
1|2016-01-01 16:16:16|Hallo Willi|2|1
2|2016-01-11 13:46:03|Selber hallo|1|2
3|2016-01-11 13:46:07|Wie geht es?|3|2
4|2016-01-11 13:46:10|Gut und selber|2|3
```

Im nächsten Schritt wollen wir nur die gesendeten Nachrichten der Person mit der id 2 ausgeben. Hierzu ist es möglich in der SELECT-Anweisung mittels einer WHERE-Klausel Einschränkungen zu definieren:

```
sqlite> SELECT zeit, text FROM nachrichten WHERE sender_id=2;
2016-01-01 16:16:16|Hallo Willi
2016-01-11 13:46:10|Gut und selber
```

In der WHERE-Klausel können auch Ungleichungen (!=,<,>,<=,>=), sowie boolesche Kombinationen dieser (AND bzw. OR als Infixoperatoren) verwendet werden.

Im nächsten Schritt möchten wir für diese Nachrichten für die Empfaenger nicht nur die ids der Personen ausgeben, sondern deren Name und Vorname. Hierzu wählen wir zunächst nicht nur Spalten aus einer Tabelle, sondern direkt aus zwei Tabellen aus:

```
sqlite> SELECT * FROM nachrichten, personen;
1|2016-01-01 16:16:16|Hallo Willi|2|1|1|Huch|Frank
1|2016-01-01 16:16:16|Hallo Willi|2|1|2|Dylus|Sandra
1|2016-01-01 16:16:16|Hallo Willi|2|1|3|Esquivel|Sandro
1|2016-01-01 16:16:16|Hallo Willi|2|1|4|Merkel|Angela
2|2016-01-11 13:46:03|Selber hallo|1|2|1|Huch|Frank
2|2016-01-11 13:46:03|Selber hallo|1|2|2|Dylus|Sandra
2|2016-01-11 13:46:03|Selber hallo|1|2|3|Esquivel|Sandro
2|2016-01-11 13:46:03|Selber hallo|1|2|4|Merkel|Angela
```

```
3|2016-01-11 13:46:07|Wie geht es?|3|2|1|Huch|Frank
3|2016-01-11 13:46:07|Wie geht es?|3|2|2|Dylus|Sandra
3|2016-01-11 13:46:07|Wie geht es?|3|2|3|Esquivel|Sandro
3|2016-01-11 13:46:07|Wie geht es?|3|2|4|Merkel|Angela
4|2016-01-11 13:46:10|Gut und selber|2|3|1|Huch|Frank
4|2016-01-11 13:46:10|Gut und selber|2|3|2|Dylus|Sandra
4|2016-01-11 13:46:10|Gut und selber|2|3|3|Esquivel|Sandro
4|2016-01-11 13:46:10|Gut und selber|2|3|4|Merkel|Angela
```

Wir erhalten also alle möglichen Kombinationen von Einträgen der einen und der anderen Tabelle. Diese entstehende Tabelle können wir nun auf die Einträge einschränken, bei denen die empfaenger_id mit der id der Tabelle personen übereinstimmt:

```
sqlite> SELECT * FROM nachrichten, personen WHERE
        sender_id=2 AND empfaenger_id=personen.id;
1|2016-01-01 16:16:16|Hallo Willi|2|1|1|Huch|Frank
4|2016-01-11 13:46:10|Gut und selber|2|3|3|Esquivel|Sandro
```

Schränken wir uns auf die interessanten Spalten ein und ordnen sie zusätzlich (alphabetisch) nach dem Veranstaltungstitel erhalten wir:

```
sqlite> SELECT nachrichten.zeit, nachrichten.text,
        personen.vorname, personen.name FROM
        nachrichten, personen WHERE
        sender_id=2 AND empfaenger_id=personen.id;
2016-01-01 16:16:16|Hallo Willi|Frank|Huch
2016-01-11 13:46:10|Gut und selber|Sandro|Esquivel
```

Ein Nachteil des Selektierens aus zwei Tabellen erkennen wir an der Anfrage

```
SELECT * FROM veranstaltung, dozent;
```

Es werden jeweils alle Einträge beider Tabellen mit einander kombiniert, was zu einer sehr großen Zwischentabelle führt und bei großen Tabellen entsprechend ineffizient werden kann. Eine Alternative stellen sogenannte Joins dar, bei denen zwei Tabellen gemäß eines Kriteriums kombiniert werden können. In unserem Beispiel kann eine effizientere Abfrage wie folgt aussehen:

```
sqlite> SELECT nachrichten.zeit, nachrichten.text,
        personen.vorname, personen.name FROM
        nachrichten INNER JOIN personen ON
        sender_id=2 AND empfaenger_id=personen.id;
```

Diese Optimierung ist aber in den meisten Fällen nicht notwendig und kann automatisch vom Datenbanksystem auch bei der ersten Anfrage hergeleitet werden. Allerdings sind viele SQL-Programmierer der Meinung, dass die Formulierung mit Hilfe eines Joins klarer ist. Deshalb werden viele Anfragen auch mit Hilfe von Joins definiert und dieses Konstrukt soll hier der Vollständigkeit halber präsentiert werden.

Als letztes SQL-Konstrukt lernen wir noch die DELETE-Anweisung kennen, mit welcher wir auch Tabelleneinträge löschen können. Als Beispiel wollen wir Frau Merkel aus der Personen-Tabelle löschen:

```
DELETE FROM personen WHERE name="Merkel";
```

Danach kommt Frau Merkel zwar nicht mehr in der Personen-Tabele vor, aber es gibt natürlich noch Veranstaltungen, welche auf sie verweisen. Diese können wir aber genauso einfach löschen:

```
DELETE FROM veranstaltung WHERE veranstaltung.dozenten_id=2;
```

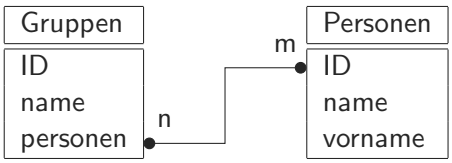
Um mit Datenbanken auch praktisch arbeiten zu können, haben wir auf der Webseite der Vorlesung eine Bibliothek (netter Weise erstellt von Stefan Exner) zur Verfügung gestellt, welche eine einfache Anbindung von Ruby an SQLite we ermöglicht. Mit ihr können SQL-Befehle in Ruby ausgeführt werden. Liefert eine SQL-Befehl (wie z.B. der SELECT-Befehl) eine Tabelle als Ergebnis, wird diese in Form eines zweidimensionalen Arrays zurück geliefert. Die Verwendung sollte mit dem folgenden Beispielprogramm klar werden:

```
require_relative("sqlite_connector")
# Anbindungsmodul für sqlite3.
# Datei sqlite_connector.rb sollte im gleichen Verzeichnis liegen

use_database("chat") do |db|
  db.execute("CREATE TABLE personen "+
                    "(id INTEGER PRIMARY KEY AUTOINCREMENT,"+
                    "name TEXT, vorname TEXT);")
  ...
  db.execute('INSERT INTO personen (id, name, vorname) '+
               'VALUES (1,"Huch", "Frank");')
  ...
  dozenten = db.execute('SELECT * FROM personen;')
  p (dozenten)  # Gibt die Tabelle der aktuellen
                # Dozenten als Array aus
end
```

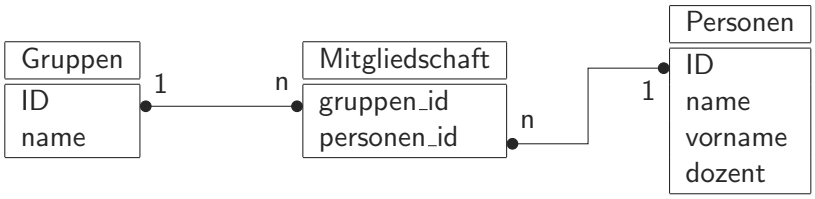
In der Vorlesung werden wir die chat-Anwendung von csv auf SQL umstellen.

Als nächstes wollen wir die chat-Anwednung noch erweitern. Hierzu wollen wir auch Gruppen-Chats in das System aufnehmen und die Definition von Gruppen modellieren. Eine Gruppe besteht (in der Regel) aus mehreren Personen. Allerdings können Personen auch in mehreren Gruppen Mitglied sein. Zwischen Personen und Gruppen besteht also ein n:m-Beziehung, was wir wir im Datenbank-Schema wie folgt notieren können:




Wie können wir eine solche Relation nun in der Datenbank abbilden. Ein erster Ansatz wäre die Verwendung einer Liste von Veranstaltungen bei jedem Studierenden. Solche Strukturen lassen sich in tabellenbasierten Datenbanken aber nur schwer realisieren, weshalb man in der Regel eine andere Idee verfolgt.

Wir führen eine separate Tabelle ein, welche genau die n:m-Beziehung repräsentiert. Diese Tabelle könnte man beispielsweise Gruppenmitgliedschaft nennen. In ihr stehen Kombinationen aus Gruppen-IDs und Personen-IDs (also der beiden Schlüsselwerte, der an der n:m-Beziehung beteiligten Tabellen).




In der Tabelle Mitgliedschaft speichern wir also einfach Paare von Gruppen-IDs und Personen-IDs. Es ist nicht sinnvoll, dass eine Person mehrmals Mitglied in der gleichen Gruppe ist, weshalb man dann beim Füllen dieser Tabelle darauf achten sollte, dass es keine doppelten Einträge gibt. Gleichzeitig ist aber die Kombination aus Gruppen-ID und Personen-ID für jede Mitgliedschaft eindeutig, so dass wir beide Attribute in Kombination als sogenanten Verbundschlüssel nutzen können und nicht extra eine ID eintragen müssen.