Funktionale und objektorientierte Programmierkonzepte Version 1.0

Fabian Damken (fabian.damken@stud.tu-darmstadt.de)

TECHNISCHE
UNIVERSITÄT
DARMSTADT

Inhaltsverzeichnis

1 Einführung **3**
1.1 Aufbau . . . 3

2 Abstrakte Konzepte **4**
2.1 Programmierparadigmen . . . 4
2.1.1 Deklarativ . . . 4
2.1.2 Funktional . . . 4
2.1.3 Imperativ . . . 5
2.1.4 Objektorientiert . . . 5
2.1.5 Abgrenzung Funktional ⟷ Imperativ . . . 5
2.2 Lexikalische Bestandteile . . . 6
2.2.1 Datentypen . . . 6
2.2.2 Literale . . . 7
2.2.3 Schlüsselwörter . . . 8
2.2.4 Bezeichner . . . 8
2.2.5 Operatoren . . . 8
2.2.6 Strukturierung von Quellcodedateien . . . 8
2.3 Anweisungen . . . 9
2.3.1 Variablen und Konstanten . . . 9
2.3.2 Zuweisungen . . . 10
2.3.3 Nutzung von Methoden . . . 10
2.3.4 Operatoren . . . 10
2.3.5 Links-/Rechtsauswertung . . . 12
2.3.6 Seiteneffekte . . . 13
2.4 Kontrollstrukturen . . . 13
2.4.1 Verzweigungen . . . 13
2.4.2 Schleifen . . . 14
2.5 Methoden/Funktionen . . . 15
2.5.1 Aufbau . . . 15
2.5.2 Verträge . . . 16
2.5.3 Rekursion . . . 17
2.5.4 Überladen . . . 17
2.5.5 Überschreiben . . . 18
2.6 Scoping . . . 18
2.7 Fehlerbehandlung . . . 19
2.7.1 Result Code . . . 19
2.7.2 Exceptions . . . 20
2.8 Generische Programmierung . . . 22
2.9 Datenstrukturen . . . 23
2.9.1 Arrays, Listen, Mengen . . . 23
2.9.2 Map . . . 25
2.10 I/O (Input/Output) . . . 25
2.10.1 Allgemeiner Aufbau . . . 26
2.11 Multithreading und parallele Verarbeitung . . . 26
2.11.1 Thread . . . 27

2.11.2 Parallelisierung . . . 27
2.11.3 Beispiel: Window Manager . . . 27
2.12 GUI (Graphical User Interface) . . . 28
2.13 Dokumentation . . . 28
2.13.1 Verträge . . . 28
2.13.2 Beispiel . . . 29
2.14 Testen . . . 31

3 Racket **33**
3.1 Lexikalische Bestandteile . . . 33
3.1.1 Datentypen . . . 33
3.1.2 Literale . . . 34
3.1.3 Bezeichner und Konventionen . . . 34
3.1.4 Strukturierung des Codes . . . 35
3.2 Anweisungen . . . 35
3.2.1 Funktionsaufrufe . . . 35
3.2.2 Konstanten . . . 36
3.2.3 „Operatoren“ . . . 36
3.2.4 Abfragen/Vergleiche . . . 36
3.3 Kontrollstrukturen . . . 37
3.3.1 If . . . 38
3.3.2 Cond . . . 38
3.4 Funktionen . . . 39
3.4.1 Bestandteile . . . 39
3.4.2 Verträge . . . 39
3.4.3 Rekursion . . . 39
3.5 Fehlerbehandlung . . . 40
3.5.1 Result Codes . . . 40
3.5.2 Errors . . . 40
3.6 Datenstrukturen . . . 41
3.6.1 Listen . . . 41
3.6.2 Structs . . . 42
3.7 Funktionen höherer Ordnung . . . 43
3.7.1 Lambdas . . . 43
3.7.2 Funktionen höherer Ordnung . . . 43
3.7.3 Funktionen als Daten . . . 44
3.7.4 Beispiele . . . 44
3.8 Dokumentation . . . 45
3.8.1 Verträge . . . 45
3.8.2 Funktionsdokumentation . . . 45
3.9 Testen . . . 46
3.10 Zusammenfassung . . . 46

4 Java **49**

5 Abstraktion **50**

1 Einführung

Dieses Skript ist Vorlesungs- und Übungsbegleitend zu der Veranstaltung „Funktionale und objektorientierte Programmierkonzepte“ zu verstehen. Es wird im Laufe des Semesters mit weiteren Kapiteln ergänzt werden, in dieser ersten Version ist nur der gesamte funktionale Teil zu Racket vollständig vorhanden. In den nächsten Wochen werden Kapitel zu Java und objektorientierter Programmierung folgen.

Ich wünsche viel Spaß [1] beim Lesen und viel Erfolg für die Klausur!

1.1 Aufbau

Dieses Skript ist in folgende Kapitel gegliedert:

2 Abstrakte Konzepte
In diesem Kapitel werden abstrakte Konzepte der Programmierung eingeführt, d.h. es wird über keine Programmiersprache an sich gesprochen.

3 Racket
Dieses Kapitel führt in die funktionale (2.1.3) Programmiersprache Racket ein, indem die im vorherigen Kapitel eingeführten Konzepte auf die Sprache angewendet werden.

4 Java
Ebenso wir im Kapitel über Racket, nur werden hier die Konzepte auf die objektorientierte (2.1.4) und imperative (2.1.3) Programmiersprache Java angewendet.

5 Abstraktion
Dieses Kapitel behandelt unterschiedliche Methodiken der Abstraktion, wie sie in funktionaler und objektorientierter Programmierung eingesetzt werden.

[1] Spaß, Spaß, Spaß

2 Abstrakte Konzepte

Dieses Kapitel führt ein in die abstrakten Konzepte, welche hinter eine Programmiersprache stehen.

Da sich nicht alle Konzepte auf alle Programmierparadigmen [1] anwenden lassen, ist jeder Abschnitt mit

gekennzeichnet, je nachdem, auf welches Paradigma sich das vorgestellte Konzept anwenden lässt. Die Markierungen werden am rechten Rand angebracht sein, sodass diese leicht zu finden sind.

2.1 Programmierparadigmen

2.1.1 Deklarativ

Dieser Abschnitt führt das Konzept von deklarativen Programmiersprachen ein.

Der der Deklarativen Programmierung steht die Beschreibung des Problems im Vordergrund, die Lösung wird hier meist automatisiert gefunden.

Es steht somit im Vordergrund, *welches* Problem gelöst werden soll und nicht *wie* ein Problem gelöst werden soll. Hierdurch ist eine genaue Trennung von Problem und Implementierung möglich, was bei imperativen Programmiersprachen (2.1.3) gar nicht oder zumindest nicht trivial möglich ist.

Das Paradigma der deklarativen Programmierung kann in weitere unterteilt werden, beispielsweise in funktionale (2.1.2) und logische Sprachen. Logische Sprachen werden hier nicht weiter ausgeführt.

Beispiele

- SQL, Cypher (Abfragesprachen)
- Lisp, Racket, Haskell (Funktionale Sprachen)
- Prolog (Logische Sprache)

2.1.2 Funktional

Dieser Abschnitt führt das Konzept von funktionalen Programmiersprachen ein.

Funktionale Programmiersprache sind Ausarbeitungen von deklarativen Sprachen (2.1.1), bei denen ebenfalls die Beschreibung des Problems im Vordergrund steht. Sie werden oftmals zur Beschreibung von mathematischen Problem verwendet.

In diesen Sprachen wir auf Konstrukte wie Schleifen (2.4.2) und Variablen (2.3.1) verzichtet, wodurch Seiteneffekte (beispielsweise das Überschreiben von Zustandsvariablen) verhindert werden und die Implementierung zur Lösung eines Problems robuster wird.

Zur Abgrenzung von funktionalen Sprachen zu imperativen Sprachen siehe 2.1.5.

[1] Siehe 2.1

Beispiele

- Lisp
- Racket
- Haskell

2.1.3 Imperativ

Imperative Programmiersprachen sehen vor, dass der Entwickler beschreibt, *wie* ein Problem zu lösen ist, wobei die Beschreibung des eigentlichen Problems (das „*Was*“) fallen gelassen wird. Ein Programm besteht „aus einer Folge von Anweisungen [...], die vorgeben, in welcher Reihenfolge was vom Computer getan werden soll“. [BJ05]

Im Gegensatz zu deklarativen und funktionalen Sprachen ist die Korrektheit eines Algorithmus weniger offensichtlich und es werden Kontrollstrukturen wie Schleifen (2.4.2) und Variablen (2.3.1) eingeführt.

Zur Abgrenzung von imperativen und funktionalen Sprachen siehe 2.1.5.

Beispiele

- Java
- C/C++
- Assembler

2.1.4 Objektorientiert

Dieser Abschnitt führt das Konzept von objektorientierten Programmiersprachen ein.

Bei der objektorientierten Programmierung (OOP) werden reale Strukturen, sogenannte Objekte in der Software abgebildet. Die Architektur der Software wird somit an bestehenden Systemen der Wirklichkeit abgebildet und erlaubt den meisten Entwickler*innen einen einfachen Zugang zu der Software, da die Wirklichkeit repräsentiert wird. Ein Programm besteht besteht aus Anweisungen, welche vorgeben, was der Computer abarbeiten soll und in welcher Reihenfolge. Somit sind (die meisten) objektorientierten Programmiersprachen ebenfalls imperativ (2.1.3).

Durch die Vermischung mit imperativen und funktionalen Paradigmen lassen sich objektorientierte Sprachen nicht hinreichend von ersteren Abgrenzen, da letztere meistens auch Teile der imperativen und funktionalen Paradigmen beinhalten.

Beispiele

- Java
- C++
- Python

2.1.5 Abgrenzung Funktional ⟷ Imperativ

Die Abgrenzung von funktionalen und imperativen Sprachen lässt sich am besten anhand eines Beispiels erläutern:

Gegeben sei das mathematische Problem, die Fakultät einer beliebigen natürlichen Zahl $n \in \mathbb{N}_0$ zu bestimmen. Mathematisch wird das Problem wie folgt rekursiv definiert:

$$n! = f(n) = \begin{cases} 1 & \text{falls } \mathrm{n} = 0 \\ n \cdot f(n-1) & \text{falls } \mathrm{n} > 0 \end{cases}$$

In einer (fiktionalen) funktionalen Sprache kann das Problem folgendermaßen implementiert werden:

```
1 f(0) := 1
2 f(n) := n * f(n - 1)
```

Abbildung 2.1: Funktionale Implementierung der Fakultät

In einer (ebenfalls fiktionalen) imperativen Sprache kann das Problem folgendermaßen implementiert werden:

```
1 function f(n)
2     num = 1
3     for i in 1..n
4         num = num * i
5     endloop
6 endfunction
```

Abbildung 2.2: Imperative Implementierung der Fakultät

2.2 Lexikalische Bestandteile

Funktional

Imperativ

Objektorientiert

Der lexikalische Bestandteil einer Programmiersprache kann auch als „Lexikon“ der Programmiersprache aufgefasst werden, welches definiert, welche Zeichen und Wörter innerhalb eines Programms verwendet werden können und welche Bedeutung diese haben. Hier entspringt auch der Name „Lexikalische Bestandteile“.

2.2.1 Datentypen

Funktional

Imperativ

Objektorientiert

Dieser Abschnitt führt das Konzept von Datentypen ein.
Wir schauen uns in diesem Kontext zuerst die Datentypen an, wodurch festgelegt wird, welche Art von Daten wir mit der Programmiersprache verarbeiten können.
Um eine Intuition dafür zu bekommen, was ein Datentyp ist folgendes Szenario: Ein Verkehrsverbund möchte Daten über seine Straßenbahnen ablegen, genauer die Start- und Endstation einer Straßenbahn („Hauptbahnhof“, bzw. „Willy-Brandt-Platz“), die Länge der Gesamtstrecke in Kilometern (ca. 2km),

den Fahrkartenpreis (2,15 €) und einen Schalter, ob die Straßenbahn gerade in Bewegung ist oder nicht.
Auffällig ist, dass sich die Arten der Daten unterscheiden:

- die Start- und Endstation der Straßenbahn ist ein Text,
- die Länge der Strecke ist eine Zahl,
- der Fahrkartenpreis ist eine Kommazahl und
- der Schalter ist ein Wahrheitswert (wahr/falsch).

Hierbei haben wir schon drei typische Datentypen in der Programmierung gefunden:

- Zeichenketten („String“),
- Ganzzahlen („Integer“/„Int“),
- Kommazahlen (genauer: Fließkommazahlen, da sich das Komma an jeder Stelle befinden kann) („Float“) und
- Wahrheitswerte („Boolean“/„Bool“).

Auch werden häufig einzelne Zeichen („Character“/„Char“) verwendet, aus welchen ein String wiederum besteht.
Abschließend bedeutet dies, dass durch einen Datentyp beschrieben wird, welche Form (Typ) die abgelegten Daten haben und später, wie diese im Speicher abgelegt werden.

2.2.2 Literale

Funktional
Imperativ
Objektorientiert

Dieser Abschnitt führt das Konzept von Literalen ein.
Als „Literale“ werden Werte bezeichnet, welche nicht über Variablen oder Konstanten (siehe 2.3.1) verarbeitet werden, sondern die direkt im Quellcode des Programms stehen (das heißt „hardcoded“ (festgeschrieben) sind).
Wir behandeln die Syntax der Erstellung und Nutzung von Literalen hier nicht weiter, da sich dies von Sprache zu Sprache stark unterscheidet. Die spezielle Syntax für Racket und Java werden wir in jeweiligen Abschnitten behandeln.

Escape-Sequenzen
Schweifen wir nochmals kurz ab zu den Literalen und betrachten sogenannte „Escape-Sequenzen“. Eine Escape-Sequenz ist eine bestimmte Abfolge von Zeichen, die bei der Ausführung durch andere Zeichen ersetzt werden.
Beispiel: Bei einer Zeichenkette, die mit einfachen Hochkommata eingeschlossen wird, ist nicht zu unterscheiden ob `''''` einen oder zwei Zeichenketten darstellt. Meistens wird nun der *Backslash* (\) genutzt, um zu sagen „das nächste Zeichen ist kein Steuerzeichen, sondern ein Buchstabe“: `'\'\''`.
Nun ist klar, dass es sich um eine Zeichenkette handelt, die aus zwei Hochkommata besteht.
Ähnlich wird es mit Zeilenumbrüchen gehandhabt. Diese werden meistens als `\n` dargestellt (wobei das „n“ für „new line“ steht).
Die genauen Escape-Sequenzen und ob diese überhaupt nötig sind hängt natürlich von der Programmiersprache ab und wir schauen es uns somit erneut an.

2.2.3 Schlüsselwörter

Funktional

Imperativ

Objektorientiert

Dieser Abschnitt führt das Konzept von Schlüsselwörtern ein.
Schlüsselwörter („Keywords“) werden in Programmiersprachen dazu eingesetzt, bestimmte Funktionen zu kennzeichnen. Sie beschreiben ganz bestimmte Zeichenfolgen (meist nur Text), welche meist nicht in einen anderen Kontexten verwendet werden dürfen (wir werden dies weiter behandeln im Abschnitt zu Einschränkungen von Namensgebungen in Racket und Java).
Die Funktionalität und die Verfügbarkeit von Schlüsselwörtern ist von der Programmiersprache abhängig und wir werden diese weiter in den entsprechenden Abschnitten im Racket und Java Teil betrachten.

2.2.4 Bezeichner

Funktional

Imperativ

Objektorientiert

Dieser Abschnitt führt das Konzept von Bezeichnern ein.
Wie auch in der realen Welt müssen wir auch in der Programmierung Namen für die Elemente unseres Programms (beispielsweise Konstanten oder Variablen (siehe 2.3.1)) Namen finden. Als Beispiel aus der Welt sei der Name „Streckenkilometer“ gegeben, um bei dem Beispiels aus dem Abschnitt über Datentypen (2.2.1) zu bleiben. Innerhalb von Programmen nennen wir solche Namen „Bezeichner“, für die auch besondere Einschränkungen (Beispiel: „Ein Bezeichner darf nicht mit einer Ziffer beginnen.“) gelten können. Da letztere natürlich von der Programmiersprache abhängig sind, werden wir dies in jeweiligen Abschnitten von Racket und Java beleuchten.

2.2.5 Operatoren

Funktional

Imperativ

Objektorientiert

Dieser Abschnitt führt das Konzept von Operatoren ein.
Ein Operator beschreibt eine Aktion auf mindestens einem Datensatz.
Ein Beispiel für einen Operator ist die Addition, welche 2 oder mehr Zahlen summiert. Wie wir einen solchen Operator in den konkreten Sprachen nutzen, hängt von der Sprache ab. Es gibt allerdings eine Operatoren, die (zumindest von der Bedeutung her) in allen Programmiersprachen vorhanden sind, beispielsweise die Addition. Diese Operationen werden im Abschnitt 2.3.4 beschrieben.

2.2.6 Strukturierung von Quellcodedateien

Funktional

Imperativ

Objektorientiert

Dieser Abschnitt führt das Konzept von Strukturierung von Quellcodedateien ein.
Da mit steigender Größe des Projektes in den meisten Fällen auch die Anzahl der Quellcodedateien steigt, muss dieser Code strukturiert werden. In den meisten Sprachen stehen hier Mechanismen zur Verfügung, um den Quellcode in eine logische Struktur zu bringen, welche nicht unbedingt physikalisch abgebildet sein muss. Beispielsweise stehen hier in C++ Namespaces und in Java Packages zur Verfügung. Letztere werden wir im Abschnitt zu Java behandeln.
Der Unterschied einer logischen und einer physikalischen Struktur besteht darin, dass ersteres nur in den Köpfen der Menschen und (meistens) auch im Quellcode verankert ist, während physikalische Strukturen direkt im Speicher abgebildet werden (zum Beispiel in Form von Verzeichnissen).

2.3 Anweisungen

Funktional

Imperativ

Objektorientiert

Nach Betrachtung des langweiligen Teil der Programmierung, den lexikalischen Bestandteilen, wenden wir und nun den Anweisungen zu, die unserem Programm Leben einhauchen. Die schreiben dem Computer vor, welche Aufgaben er abzuarbeiten hat und in welcher Reihenfolge.

2.3.1 Variablen und Konstanten

Imperativ

Objektorientiert

Dieser Abschnitt führt das Konzept von Variablen Konstanten ein.
Stellen wir uns vor, wir wollen das Durchschnittsalter aller Studierenden am Fachbereich Informatik berechnen. Zur Einfachheit nehmen wir hier an, es gibt nur 10 Studierende, welche die Alter $21, 41, 27, 18, 20, 24, 30, 17, 29, 25$ haben. Zur Berechnung des Durchschnitts müssen wir nun alle Zahlen aufsummieren, uns das Ergebnis merken und durch die Anzahl an Studierenden dividieren. Wir merken hier schnell, das wir uns „das Ergebnis merken“ müssen, da wir nicht mit so vielen Zahlen zur gleichen Zeit rechnen können.
Das gleiche Phänomen tritt auch auf, wenn wir obigen Algorithmus programmieren wollen: Wir müssen uns Daten zwischenspeichern.
Dies ist genau der Punkt, an dem Variablen ins Spiel kommen: Diese stellen einen kleinen, modifizierbaren, Speicher dar, dem eine Aufgabe und ein Datentyp zugeordnet ist. In unserem Fall ist die Aufgabe *das Halten des Zwischenergebnisses* und der Datentyp *Integer*. Um die Aufgabe erkenntlich zu machen, geben wir der Variablen einen Namen: „Gesamtalter“.
Um das alles Zusammen zu fassen: Eine Variable ist ein benannter Zwischenspeicher innerhalb eines Programms, an dem wir Daten eines bestimmten Datentyps speichern können und die Werte verändern. Selbstverständlich können wir auf die gespeicherten Werte auch zugreifen, sonst wäre das Speichern sinnlos [2].
Eine Konstante ist im Prinzip das gleiche wie eine Variable, nur ist sie nicht änderbar (sie ist „konstant“). Dies ist sinnvoll, wenn wir den gleichen Wert häufig im Programm verwenden, diesen aber nicht immer Tippen möchten. Ein prominentes Beispiel hierfür ist die Kreiszahl $\pi \approx 3,141592653589793$, welche in fast allen Sprachen bereits als Konstante vorliegt.

[2] In Java gibt es hier eine Ausnahme, eine sogenannte „Phantom Reference“. Für mehr Informationen siehe https://docs.oracle.com/javase/10/docs/api/java/lang/ref/PhantomReference.html

Begriffe

- Wenn eine Variable erstellt wird, das heißt der Typ und der Name wird festgelegt, wird dies [Deklaration] genannt.
- Die erste Festlegung, welchen Wert eine Variable am Anfang haben soll, wird *Initialisierung* genannt.
  Bei Konstanten ist dies dies einzige Wertfestlegung, die stattfinden kann.

2.3.2 Zuweisungen

Imperativ

Objektorientiert

Dieser Abschnitt führt das Konzept von Zuweisungen ein.
Wenn der Wert einer Variablen festgelegt wird, wird dies *Zuweisung* genannt. Wie im Abschnitt zu Variablen und Konstanten (2.3.1) bereits erwähnt, kann einer Konstanten nur einmalig ein Wert zugewiesen werden.
Näheres zu Zuweisungen wird im Abschnitt Links-/Rechtsauswertung (2.3.5) behandelt.

2.3.3 Nutzung von Methoden

Funktional

Imperativ

Objektorientiert

Dieser Abschnitt führt das Konzept Methoden ein.
Fürs erste reicht es uns hier zu wissen, dass eine Methode ein weg ist, um Code zu deduplizieren an zentraler Stelle zu halten. Näheres über Methoden werden wir im Abschnitt zu Methoden und Funktionen (2.5) behandeln.
Wir werden oftmals feststellen, dass einige Dinge schwierig zu implementieren sind. Zum Glück haben die meisten Programmiersprachen eine sogenannte *Standardbibliothek*, welche viele Methoden zur Verfügung stellt, die Standardaufgaben implementieren (beispielsweise die Quadratwurzel). Um diese Methoden zu nutzen, müssen die Methoden aufgerufen werden. Dies ist zu Vergleichen mit dem Aufruf einer Funktion in der Mathematik, beispielsweise der Quadratwurzelfunktion
($\sqrt{x} := y \in \mathbb{R}_+ : y \cdot y = x$). Wird übergeben einer Methode *Parameter*, die Methode bearbeitet diese irgendwie und liefert uns ein *Ergebnis* (Eingabe-Verarbeitung-Ausgabe, EVA-Prinzip). Im Fall von der Quadratwurzel übergeben wir der Funktion als Parameter eine Zahl $x \in \mathbb{R}_+$ und kriegen ein solches Ergebnis, sodass $x = \sqrt{x} \cdot \sqrt{x}$ gilt (das dies in einigen Fällen nicht oder nicht vollständig lösbar ist, behandeln wir hier nicht weiter).

2.3.4 Operatoren

Funktional

Imperativ

Objektorientiert

Dieser Abschnitt führt das Konzept von Operatoren ein.
Wir betrachten nun erneut Operatoren, welche wir bereits bei den lexikalischen Bestandteilen behandelt haben (siehe 2.2.5).

In diesem Abschnitt behandeln wir nun einige Grundbegriffe, welche im Zusammenhang mit Operatoren immer wieder verwendet werden und schauen uns einige grundlegende Operatoren an, welche in annähernd allen Programmiersprachen vorhanden sind [3].

Arithmetische Operatoren

Oftmals vorhandene arithmetische Operatoren sind „Addition“, „Subtraktion“, „Multiplikation“, „Division“ und „Modulo“. Hierbei sind auch in der Programmierung sämtliche aus der Mathematik bekannte Gesetze und Axiome anwendbar. Ein Beispiel hierfür ist die Kommutativität der Addition.
Der Operator „Modulo“ gibt uns den Wert des Restes bei einer Division zurück (Beispiel: Der Ausdruck „Rechne 5 modulo 3“ gibt uns den Wert 2, da: $1 \cdot 3 + 2 = 5$).
Mathematisch kann man den Modulo-Operator wie folgt definieren:

$$x \text{ modulo } y := x - \lfloor \frac{x}{y} \rfloor$$

Logische Operatoren

Auch vorhanden sind logische Operatoren wie „AND“ (logisches Und), „OR“ (logisches Oder), „NOT“ (logische Negation) und „XOR“ (logisches exklusives Oder). Diese arbeiten auf Wahrheitswerten und ergeben folgende Wahrheitstafeln [4]:

A	B	A AND B	A OR B	NOT A	NOT B	A XOR B
𝔣	𝔣	𝔣	𝔣	𝔴	𝔴	𝔣
𝔣	𝔴	𝔣	𝔴	𝔴	𝔣	𝔴
𝔴	𝔣	𝔣	𝔴	𝔣	𝔴	𝔴
𝔴	𝔴	𝔴	𝔴	𝔣	𝔣	𝔣

Tabelle 2.1: Wahrheitstafel für „and“, „or“, „not“ und „xor“

> **Warnung:** Das logische Oder ist genau dann war, wenn **mindestens ein Parameter** wahr ist. Das in der Prosa übliche oder (entweder-oder) ist im *exklusiven oder* („xor“) implementiert und wird genau dann war, wenn **genau ein Parameter** wahr ist (in anderen Worten: Wenn die Parameter unterschiedlich sind).

Auch für die logischen Operatoren gilt, dass Gesetze und Axiome aus der Logik auch in der Programmierung anwendbar sind (beispielsweise die De Morganschen Gesetze).

Bitlogische Operatoren

Bitlogische Operatoren arbeiten ähnlich zu den logischen Operatoren, können aber auf Zahlen operieren. Hierbei wird jedes Bit einer Zahl einzeln betrachtet, sodass wieder mit *Wahr* (`1`) und *Falsch* (`0`) gearbeitet werden kann. Hier sind üblicherweise ebenfalls die üblichen logischen Operatoren „AND“, „OR“, „NOT“ und „XOR“ verfügbar.

[3] Ausnahmen bilden hier manche esoterische Programmiersprachen wie beispielsweise Brainfuck. Siehe hierzu https://de.wikipedia.org/wiki/Brainfuck

[4] Wahrheitstafeln beschreiben, bei welchen durch welche Operationen welche Ergebnisse vorliegen

Wir betrachtet nun einige Beispiele, welche allesamt auf den Zahlen $\alpha = 1100_2 = 12_{10}$ und $\beta = 1010_2 = 10_{10}$ operieren, dabei ist sowohl das Ergebnis in Binär (tiefgestellte 2) und in Dezimal (tiefgestellte 10) angegeben:

	Binär	Dezimal
α AND β	1000_2	8_{10}
α OR β	1110_2	14_{10}
NOT α	0011_2	3_{10}
NOT β	0101_2	5_{10}
α XOR β	0110_2	6_{10}

Tabelle 2.2: Ergebnistafel für bitlogische Operationen

Bindungsstärke

Da meistens mehrere Operatoren innerhalb eines Ausdrucks verwendet werden, ist es wichtig zu wissen, in welcher Reihenfolge die Operatoren ausgeführt werden. Diese Reihenfolge wird „Auswertungsreihenfolge“ genannt und basiert auf der „Bindungsstärke“ von Operatoren. Ein Operator mit einer hohen Bindungsstärke wird hierbei vor einem Operatoren mit einer geringeren Bindungsstärke ausgeführt.
Schauen wir uns zur Veranschaulichung das simpelste Beispiel an, welches wir alle aus der Schulmathematik kennen: „Punkt vor Strich“.
Mit dieser Regel wird festgelegt, wie ein Ausdruck $5 + 3 \cdot 7$ ausgewertet wird, nämlich:

$$5 + 3 \cdot 7 = 5 + 21 = 26$$

und nicht wie folgt, wenn „Strich vor Punkt“ gelten würde:

$$5 + 3 \cdot 7 = 8 \cdot 7 = 56$$

In anderen Worten: Die Operatoren „Multiplikation“ und „Division“ haben eine höhere Bindungsstärke

2.3.5 Links-/Rechtsauswertung

Imperativ

Objektorientiert

Dieser Abschnitt führt das Konzept von Links- und Rechtsauswertungen ein.
Eine Auswertung findet immer dann statt, wenn der Wert eines Ausdrucks bestimmt wird.
Beispielsweise sagen wir, der Ausdruck $1 + 3$ wertet zu 4 aus.
Wird dieser Wert (das Ergebnis des Ausdrucks) anschließend einer Variable zugewiesen, wird von Links- und Rechtsauswertungen gesprochen. Die Auswertung des zuzuweisenden Wertes wird dabei *Rechtsauswertung* genannt. Der Ausdruck, der bestimmt, wem/was der Wert zugewiesen werden soll, wird *Linksauswertung* genannt.
Die Namensgebung resultiert daher, dass in den meisten Programmiersprachen der zuzuweisende Wert rechts und die Variable links steht, Beispiel:

```
1 a[b] = a[b] + 1
```

Hierbei wertet der Linksausdruck (`a[b]`) zu einer Position im Speicher aus, an der das Element gespeichert werden soll (in diesem Listing werden Arrays verwendet, diese werden erst im Abschnitt über Datenstrukturen (2.9) eingeführt), und der Rechtsausdruck (`a[b] + 1`) zu dem Wert, welcher gespeichert werden soll.

2.3.6 Seiteneffekte

Imperativ

Objektorientiert

Dieser Abschnitt führt das Konzept von Seiteneffekten ein.
Was bei funktionalen Sprachen verhindert wird, tritt bei imperativen Sprachen häufig auf: Seiteneffekte. Als Seiteneffekt wird bezeichnet, wenn sich der Zustand von etwas ändert, beispielsweise von einer Variablen. Wir sehen hier bereits, dass Seiteneffekte integraler Bestandteil von imperativen Sprachen sind (in funktionalen Sprachen nicht, da es hier kein Konzept von Variablen gibt). Seiteneffekte werden dann problematisch, wenn sich der Zustand einer Variablen verändert, ohne dass wir direkt Einfluss darauf nehmen können (zum Beispiel wenn mehrere Codeblöcke parallel zueinander ablaufen, siehe 2.11).
Wir werden weitere Probleme mit Seiteneffekten im späteren Kapitel über Java behandeln.

2.4 Kontrollstrukturen

Funktional

Imperativ

Objektorientiert

Dieser Abschnitt führt das Konzept von Kontrollstrukturen ein.
Bisher haben wir nur Konzepte betrachtet, die es uns ermöglichen, einen linearen und immer gleichen Ablauf des Programms zu bewerkstelligen. Aber spätestens, wenn unser Programm an irgendeiner Stelle „denken“ (wir betrachten hier keine Konzepte des Machine Learnings o.ä.) soll, müssen wir anfangen, darüber nachzudenken, wie wir Entscheidungen implementieren.
Hier kommen Kontrollstrukturen ins Spiel, welche uns erlauben

a) Entscheidungen zu treffen und

b) Codeblöcke zu wiederholen.

2.4.1 Verzweigungen

Funktional

Imperativ

Objektorientiert

Dieser Abschnitt führt das Konzept von Verzweigungen ein.
Wir beschäftigen uns nun mit Teil a) von obiger Liste: Verzweigungen, das wichtigste überhaupt, wenn es darum geht, in einem Programm Entscheidungen zu treffen. Wir werden nun den Haupttyp Typen von Verzweigungen kennen lernen: ein *if*.

If
Ein *if* ist eine einfache Verzweigung der Form „Wenn . . . gilt, dann tue . . . . Sonst tue . . . .“. In den meisten Programmiersprachen wird dies gesprochen als „**if** . . . **then** . . . **else** . . . “, was einer einfachen Übersetzung des deutschen „wenn, dann, ansonsten“ entspricht. Der sogenannte *else-Block* kann bei den meisten Sprachen auch fallengelassen werden, wird die Bedingung dann zu *Falsch* ausgewertet, so geschieht einfach nichts.

In vielen Fällen muss man allerdings mehr als einen Fall betrachten, wodurch sich *elseif-Blöcke* ergeben, die ungefähr die Form „**if** ... **then** ... **elseif** ... **then** ... **else** ... “ haben. Umgangssprachlich kann man ein else-if also als „wenn, dann, ansonsten wenn, $\cdots$, ansonsten“ ausdrücken. Es kann in den meisten Fällen beliebig viele else-if-Blöcke geben.

2.4.2 Schleifen

Imperativ

Objektorientiert

Dieser Abschnitt führt das Konzept von Schleifen ein.
Nun beschäftigen wir uns mit Teil b) aus obiger Liste: Schleifen. Spätestens, wenn wir unseren Code mehrmals Ausführen wollen und ihn zu diesem Zweck nicht einfach untereinander kopieren können (beispielsweise wenn die Ausführung von einem Parameter abhängt, dessen Wert wir während dem Schreiben noch nicht kennen), müssen wir unseren Code dynamisch beliebig oft ausführen. Dies ist zum Beispiel der Fall, wenn wir die Fakultät einer Zahl berechnen wollen:

$$n! := \prod_{i=1}^{n} i = 1 \cdot 2 \cdot \cdots \cdot (n-1) \cdot n$$

Als Grundlage für alle Schleifen dient die *while-Schleife*, bei der ein Codeblock so lange ausgeführt wird, bis eine bestimmte Bedingung zu *Falsch* auswertet. Der Code kann somit als „Solange ... tue ... “ verstanden werden und sieht in den meisten Sprachen auch ähnlich aus: „**while** ... **do**“. Als Anlehnung an die while-Schleife gibt es selten auch die until-Schleife, die genau entgegengesetzt funktioniert: Der Codeblock wird so lange ausgeführt, bis eine bestimmte Bedingung zu *Wahr* auswertet.
Damit können wir nun das obige Problem wie folgt in unserer imaginären Sprache lösen, wobei wir hier in der Variable `n` das n von oben speichern und in der Variable `x` das Ergebnis (es soll also nach der Ausführung `x` $= n!$ gelten).

```
1 x = 1
2 while n > 0
3     x = x * n
4
5     n = n - 1
6 done
```

Abbildung 2.3: Beispiel: While-Schleife

Der Code `x = x * n, n = n - 1` wird nun immer ausgeführt, solange `n` > 0 gilt. Eine Ausführung des Blocks wird „Schleifendurchlauf“ oder „Iteration“ genannt.
Da es manche Fälle gibt, in denen die gesamte Ausführung innerhalb der Schleife abgebrochen werden soll oder die aktuelle Iteration abgebrochen werden soll und mit der nächsten begonnen werden soll, gibt es meist noch die Ausdrücke „break“ und „continue“, welche ihrem Namen treu bleiben und die folgenden Funktionen erfüllen:

break Hält die gesamt Schleifenausführung an und fährt mit der ersten Zeile nach der Schleife fort.

continue Hält den aktuellen Schleifendurchlauf an und fährt mit der nächsten Iteration fort. Gibt es keine weitere Iteration, so wird die Schleife beendet.

2.5 Methoden/Funktionen

Funktional

Imperativ

Objektorientiert

Dieser Abschnitt führt das Konzept von Methoden ein.
Mit steigender Komplexität der Programme werden wir sehen, dass sich oftmals viele Stellen im Code doppelt, dreifach oder noch öfter zu finden sind. Auch wird ersichtlich, dass ein Programm, welches aus > 200 Zeilen besteht, nicht mehr übersichtlich ist.
Hier können *Methoden* helfen, welche den Code strukturieren.
Eine Methode nimmt Daten entgegen (Eingabe), verarbeitet diese (Verarbeitung) und gibt das Ergebnis aus (Ausgabe). Somit setzen Methoden das EVA-Prinzip der Informatik um und können als Blackbox betrachtet werden, die eine bestimmte Aufgabe (beispielsweise Wurzelziehen) erledigen.

2.5.1 Aufbau

Funktional

Imperativ

Objektorientiert

Wie bereits erwähnt nimmt eine Methode Daten entgegen, verarbeitet diese und gibt das Ergebnis aus (wenn eines berechnet wurde).
Die eingehenden Daten werden über *Parameter* übergeben, im *Körper* verarbeitet und mit einer *Rückgabe* zurück gegeben. Dabei entspricht bei funktionalen Sprachen eine Methode einer Mathematischen Funktion, welche genau eine Sache tut und Werte zurück gibt. Bei imperativer Programmierung ist es möglich, dass Methoden keinen Rückgabewert haben und mehrere Dinge tun können (mehrere Zeilen Code).
Schauen wir uns diese Einzelkomponenten von Methoden nochmals genauer an.

Parameter

Funktional

Imperativ

Objektorientiert

Ein *Parameter* wird von einer Methode spezifiziert (die Anzahl und der Typ der Parameter) und kann innerhalb der Methode wie eine normale Variable (oder in machen Sprachen Konstante) verwendet werden. Sie stellen die Eingabe der Daten in die Methode dar und werden vom Aufrufer übergeben.
Parameter sind auch bekannt aus der Mathematik, beispielsweise das „x“ in der Funktion

$$f : \mathbb{N} \to \mathbb{R} : \underline{x} \mapsto x^2$$

wobei „$\mathbb{N}$“ den Typ des Parameters festlegt.

Formale Parameter vs. Aktualparameter
Wir unterscheiden zwischen den *Formalen Parametern* und den *Aktualen Parametern*.

- Die *Formalen Parameter* sind diejenigen Parameter, die bei der Methodendefinition angegeben werden.
- Die *Aktualen Parameter* sind diejenigen Parameter, die bei dem Methodenaufruf angegeben werden.

Körper

Funktional
Imperativ
Objektorientiert

Der Code innerhalb einer Methode wird als *Körper* der Methode bezeichnet. Dieser nutzt die Parameter der Methode, enthält den nötigen Code für die zu erledigende Aufgabe und führt die Rückgabe aus. Er stellt den wichtigsten Teil der Methode dar, denn ohne Code wäre eine Methode sinnfrei.
Wieder als mathematische Funktion betrachtet, kann das „x^2“ als Körper der Funktion

$$f : \mathbb{N} \to \mathbb{R} : x \mapsto \underline{x^2}$$

verstanden werden.

Rückgabe

Funktional
Imperativ
Objektorientiert

Jede Methode hat eine *Rückgabe*, welche bei imperativen Sprachen auch leer sein kann (also kein expliziter Rückgabewert). Dies wird meist als *Void* (englisch für „Nichts“) bezeichnet. Der *Rückgabewert* ist der Wert, den die Methode aus den Parametern berechnet hat und den der Aufrufer erhält. In einer (statisch) typisierten Sprache wird außerdem der Rückgabetyp definiert, mit dem der Aufrufer rechnen kann. Die Rückgabe ist bei einer funktionalen Programmiersprache implizit, das heißt es wird nicht genau angegeben, an welcher Stelle ein Wert zurück gegeben wird. In einer imperativen Programmiersprache erfolgt die Rückgabe explizit und kann auch schon vor vollständigem Ablauf der Methode ausgeführt werden. In diesem Fall bricht die Ausführung der Methode ab. Dies kann zum Beispiels nützlich sein, wenn auf invalide Eingaben (beispielsweise negative Zahlen) geprüft wird.
Erneut als mathematische Funktion

$$f : \underline{\mathbb{N}} \to \mathbb{R} : x \mapsto x^2$$

betrachtet, kann „$\mathbb{R}$“ als Rückgabetyp verstanden werden. Da mathematische Funktionen funktional sind, findet keine explizite Rückgabe statt, sondern es wird einfach das Quadrat von x zurück gegeben.

2.5.2 Verträge

Funktional
Imperativ
Objektorientiert

Verträge sind Teil der Dokumentation, siehe Abschnitt 2.13.1.

2.5.3 Rekursion

Funktional

Imperativ

Objektorientiert

Dieser Abschnitt führt das Konzept von Rekursion ein.

> Um die Rekursion zu verstehen, muss man zuerst die Rekursion (Siehe 2.5.3) verstehen.

Rekursion bezeichnet ein Paradigma, bei dem sich eine Methode selbst aufruft und damit eine Schleife vermeidet. Dadurch ist es möglich, viele Probleme sehr übersichtlich zu lösen, wodurch der Code lesbarer und damit wartbarer wird. Wenn man es mit der Rekursion allerdings übertreibt, wird der Code sehr schwer verständlich.
Schauen wir uns als Beispiel eine mathematische Definition an, bei der Rekursion eingesetzt wird: Die Fakultät $n!$ einer natürlichen Zahl $n \in \mathbb{N}$:

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

Hier wird im zweiten Fall der Fallunterscheidung Rekursion eingesetzt, indem die Fakultät-Funktion erneut genutzt wird.

Rekursionsanker
Der *Rekursionsanker* bezeichnet den Teil der Funktion, der dafür zuständig ist, dass der Methodenaufruf beendet wird. Im obigen Beispiel ist dies der erste Fall der Fallunterscheidung, da dieser die Rekursion abbricht.
Es kann auch mehrere Rekursionsanker geben, die zusammen zum Ende der Rekursion führen.
Übung: In folgendem Beispiel wird die n-te Fibonacci-Zahl berechnet. Was ist in diesem Fall der Rekursionsanker?

$$\text{Fib}(n) = \begin{cases} 0 & \text{falls } n = 0 \\ 1 & \text{falls } n = 1 \\ \text{Fib}(n-1) + \text{Fib}(n-2) & \text{sonst} \end{cases}$$

Dies führt zu der Fibonacci-Folge

$$0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, 987, 1597, \cdots$$

2.5.4 Überladen

Funktional

Imperativ

Objektorientiert

In den meisten Programmiersprachen ist es möglich, eine mehrere Methoden mit dem gleichen Namen, aber unterschiedlichen Parametern, zu implementieren. Dann wird anhand der Parameter entschieden, welche Methode ausgeführt werden soll. Je nach Sprache ist nur möglich, nach Anzahl der Parameter zu unterscheiden oder auch nach Typ zu unterscheiden (Java, Kotlin, ...). Andere Sprachen unterstützen Überladung gar nicht (Racket, Python, ...).

2.5.5 Überschreiben

Objektorientiert

Im Kontext von objektorientierter Programmierung und Vererbung ist es von Zeit zu Zeit nötig, eine Implementierung einer Funktion von der Oberklasse zu ändern. Hierzu kann die gleiche Methode erneut implementiert werden, welche anschließend die vorherige Implementation ersetzt. Dies wird *Überschreiben* von Methoden genannt. Die meisten Sprachen bieten hier auch die Möglichkeit, explizit die „alte“ Implementierung aus der Oberklasse aufzurufen.

2.6 Scoping

Funktional

Imperativ

Objektorientiert

Dieser Abschnitt führt das Konzept von Scoping und Scopes ein.
Stellen wir uns vor, in einer Programmiersprache wären *alle* Variablen von *überall* zugreifbar und veränderbar. Dies würde zu Chaos führen, spätestens wenn zwei Schleifen mit dem Laufindex `i` zeitgleich ausgeführt werden.
Um Szenarien wie diese zu vermeiden, unterstützen die meisten Sprachen *Scoping*, also bestimmte Regeln, von wo aus eine Variable zugreifbar ist und wann Variablen mit dem gleichen Namen völlig unterschiedliche Werte repräsentieren können. Diese Regeln sind zumeist sehr simpel wie und legen beispielsweise fest, dass in einer Methode definierte Variablen nicht außerhalb dieser verwendet werden können. Ein Bereich, in dem eine Variable gültig ist, wird *Scope* genannt.
Schauen wir uns mit obiger Regel zuerst ein Beispiel an, bevor wir zu den grundlegenden Scoping-Typen fortfahren.

Beispiel
Wir betrachten folgendes Stück Code mit obiger Scoping-Regeln, dass unterschiedliche Methoden unterschiedliche Variablen definieren und unabhängig voneinander sind:

```
1  foo() {
2      a = "foo/a"; // Anlegen einer neuen Variable 'a' im Scope 'foo'.
3      b = "foo/b"; // Anlegen einer neuen Variable 'b' im Scope 'foo'.
4
5      print(a);    // Gibt 'foo/a' aus.
6      print(b);    // Gibt 'foo/b' aus.
7  }
8
9  bar() {
10     a = "bar/a"; // Anlegen einer neuen Variable 'a' im Scope 'bar', von
           'foo' unabhaengig.
11
12     print(a);    // Gibt 'bar/a' aus.
13     print(b);    // Schlaegt fehl, da 'b' im Scope 'bar' nicht definiert
           ist.
14 }
```

Abbildung 2.4: Scoping

Typen von Scopen

Bei imperativen Sprachen existieren die folgenden grundlegenden Arten von Scoping:

Global Alle Variablen sind global gültig und können von überall verändert werden. Dies stellt eigentlich keinen Scoping-Typ dar, da kein Scoping vorgenommen wird.

Function-Based Variablen sind nur innerhalb einer Funktion gültig und können auch nur dort verwendet werden.
Beispiele: Python, BASH

Block-Based Variablen sind nur innerhalb eines Codeblocks (beispielsweise abgetrennt durch geschweifte Klammern) gültig. Ein Codeblock kann beispielsweise mit einer Schleife, einem If, ... eingeleitet werden.
Beispiele: Java, Kotlin

Mixed Es werden unterschiedliche Scoping-Verfahren implementiert und der Entwickler entscheidet, welches er nutzen will.
Beispiele: JavaScript (eine globale Variable wird ohne Schlüsselwort, eine funktionslokale Variable mit dem Schlüsselwort `var` und eine blocklokale Variable mit dem Schlüsselwort `let` eingeleitet)

2.7 Fehlerbehandlung

Funktional

Imperativ

Objektorientiert

Dieser Abschnitt führt das Konzept von Fehlerbehandlung ein.
Während der Ausführung von Code kann es immer zu Fehler kommen, welche es zu behandeln gilt (beispielsweise bei einer Division durch 0). Hierbei stellt sich die Frage, wie wir dem Nutzer (oder dem Aufrufer einer Methode) erkenntlich machen, dass es zu einem Fehler gekommen ist.
Dabei gibt es unterschiedliche Möglichkeiten, welche sich grob wie folgt einteilen lassen:

- Exceptions und
- Result Codes.

Diese zwei Unterarten werden wir in den folgenden Abschnitten behandeln.

2.7.1 Result Code

Dieser Abschnitt führt das Konzept von Result Codes ein.
Beschäftigen wir uns zuerst mit der einfachsten Methode, Fehler anzuzeigen: Wir sagen dem Aufrufer über den Rückgabewert der Methode Bescheid, ob alles korrekt abgelaufen ist.
An einem konkreten Beispiel heißt dies:

- Szenario: Wir haben eine Methode `indexOf(val: String, el: char): int`, welche uns die Position des ersten Vorkommens von `el` in `val` zurück gibt.
  Beispiel: `indexOf(äsdfgas", 's')` gibt 1 zurück.
- Ist das Zeichen nicht in dem String vorhanden, stellt dies einen Fehler dar.

- Mit Fehlermeldung über Result Codes könnten wir nun beispielsweise `-1` zurück geben, da dies kein valider Index ist (welche immer $\geq$ `0` sein müssen).

- Damit sieht der Aufrufer, dass der Methodenaufruf schief gegangen ist und kann entsprechend reagieren.

Vorteile

- Es werden keine expliziten Verfahren zum Melden von Fehlern benötigt.

- Die genutzt Technologie wird in vielen Sprachen eingesetzt.

Nachteile

- Manchmal ist es nicht möglich, Fehler so anzuzeigen (beispielsweise wenn alle Werte gültig sind).

- Der Aufrufer muss extra prüfen und daran denken, ob und welche Codes zurück kommen könnten.

2.7.2 Exceptions

Dieser Abschnitt führt das Konzept von Exceptions ein.

Schauen wir uns nun *Exception* an, ein sehr viel mächtigeres System als Result Codes.

Die Grundidee einer Exception ist, dass die Ausführung des Codes an einer beliebigen Stelle abgebrochen wird und dem Aufrufer über einen weiteren Mechanismus (den Exceptions) aufgezeigt wird, dass es Fehler vorlag. Der Aufrufer kann den Fehler anschließend behandeln.

Das System besteht aus den folgenden Teilen, welche wir in den nächsten Abschnitten näher betrachten:

- Werfen von Exceptions und

- Fangen von Exceptions.

Werfen von Exceptions

Eine Methode, welche beispielsweise die Berechnung $\frac{a}{b}$ durchführt, muss einen Fehler auslösen, wenn $b = 0$ gilt.

Im Kontext von Exceptions wird dieses Auslösen eines Fehler *werfen* einer Exception genannt, das heißt der Code bricht ab, es wird kein Wert zurück gegeben und der Aufrufer „erhält“ den Fehler, welcher eine genauere Beschreibung enthalten kann (beispielsweise die Nachricht „MathException: Cannot divide by 0.“).

Beispiel

```
divide(int a, int b) {
    if (b == 0) {
        // Nach der folgenden Zeile wird zum Aufrufer zurueck gekehrt, dieser
        // erhaelt die Nachricht und die Ausfuehrung der Methode wird
        // abgebrochen.
        throw "MathException: Cannot divide by 0." // Werfen der Exception.
    }

    // Somit koennen wir uns nun sicher sein, dass 'b != 0' gilt und einfach mit
    // der Division forfahren.
    return /* Divisions-Algorithmus */
}
```

Abbildung 2.5: Exceptions Werfen: Beispiel

Fangen von Exceptions

Rufen wir eine Methode auf, welche eine Exception werfen kann (dies wird je nach Sprache in der Signatur der Methode dokumentiert), so müssen wir diese *fangen*. Das bedeutet, wir müssen die Exception empfangen und den Fehler behandeln (wie auch immer).

Dies geschieht zumeist mit einem Try-Catch-Konstrukt, welcher in zwei Blöcke aufgeteilt ist:

- Der *Try-Block* enthält den Code, der die Exception auslösen kann. Tritt irgendwo eine Exception auf, so bricht die Ausführung dieses Blocks ab.

- Der *Catch-Block* fängt eine mögliche Exception und wird nur ausgeführt, wenn im Try-Block ein Fehler aufgetreten ist. Sofern der Catch-Block nicht für einen Abbruch der Ausführung sorgt, wird nach seiner Ausführung einfach mit der ersten Zeile nach dem Try-Catch fortgefahren.

- In den meisten Sprachen gibt es noch einen Finally-Block, diesen werden wir aber erst im Java-Abschnitt zu Exceptions behandeln.

Beispiel

Sei wieder die Methode aus Abbildung 2.5 gegeben, nur rufen wir diese diesmal auf.

```
// Try-Catch-Konstrukt
try { // Try-Block

    // Dieser Aufruf geht noch gut, denn '2 != 0'.
    divide(4, -2)
    // Dieser Aufruf wird fehlschlagen und der Catch-Block wird
       ausgefuehrt.
    divide(5, 0)
    // Dieser Aufruf wird nicht mehr ausgefuehrt, da der vorherige Aufruf
    // fehlgeschlagen ist.
    divide(6, 2)
} catch (String exception) { // Catch-Block

    // Der String 'exception' enthaelt nun den Wert "MathException: Cannot
       divide
    // by 0.", welcher von der Methode divide(int, int) als Fehlermeldung
    // uebergeben wurde.
    // Wir geben den Fehler hier einfach aus und behandeln ihn nicht
       weiter.
    print(exception)
}
```

Abbildung 2.6: Exceptions Fangen: Beispiel

Exception-Typen

Es wird im allgemeinen zwischen den folgenden Exception-Typen unterschieden:

Geprüft Diese Exceptions müssen von dem Aufrufer gefangen und behandelt oder weitergeleitet werden. Das Ignorieren der Exception führt zu einem Compiler Fehler.

Nicht Geprüft Diese Exceptions müssen nicht von dem Aufrufer gefangen werden und können ignoriert werden. Allerdings stürzt das Programm ab, sollte doch ein solcher Fehler auftreten.

> **Info:** Die Diskussion, ob man nur geprüfte Exceptions, nur ungeprüfte Exceptions oder beides verwenden sollte, ist sehr langwierig und es gibt für beide Seiten gute Argumente. Meiner Meinung nach ist die Mischform der beste Weg, da dieser am meisten Flexibilität bietet.

2.8 Generische Programmierung

Funktional

Imperativ

Objektorientiert

Dieser Abschnitt wird in den nächsten Wochen folgen.

2.9 Datenstrukturen

Funktional

Imperativ

Objektorientiert

Dieser Abschnitt führt das Konzept von Datenstrukturen ein.
Eine Datenstruktur ist ein Objekt zur Speicherung und Organisation von Daten, indem diese in einer bestimmten Art und Weise angeordnet sind und es klare Namen gibt, sodass unterschiedliche Entwickler sich über Datenstrukturen unterhalten können, ohne an eine bestimmte Programmiersprache gebunden zu sein.

> **Info:** In dieser Veranstaltung werden wir Datenstrukturen nur grob behandeln, genauer wird dies in der Veranstaltung „Algorithmen und Datenstrukturen“ behandelt.

Im allgemeinen unterscheidet man zwischen folgenden Typen von Datenstrukturen:

Indexbasiert Jedes Element innerhalb einer Datenstruktur

Nicht Indexbasiert

2.9.1 Arrays, Listen, Mengen

Funktional

Imperativ

Objektorientiert

Bevor wir uns einige Implementierungen von Arrays, Listen und Mengen anschauen, stellt sich zuerst die Frage, was das eigentlich alles ist?
Alle drei stellen eine Auflistung von Elementen eines beliebigen Typs dar und dienen dazu, beliebig viele und noch nicht zur Compile-Zeit bekannte Elemente in einer Variablen zusammenzufassen. Dabei wird unterschieden zwischen *indexbasierten* und *nicht indexbasierten* Strukturen, wobei bei ersteren jedes Element mit einem Index (einer Zahl) identifiziert werden kann, bei letzteren nicht.

Array

Dieser Abschnitt führt das Konzept von Arrays ein.
Ein Array ist eine solche Auflistung, welche in den meisten Sprachen nicht zur Laufzeit vergrößert werden kann und in einigen Sprachen (beispielsweise C) sogar schon zur Laufzeit feststehen muss.
Grob gesagt kann man sich ein Array als beschriftetes Kistensystem vorstellen, bei dem jede Kiste eine Zahl zugewiesen bekommt:

Abbildung 2.7: Datenstruktur: Array

Warnung: In annähernd allen Programmiersprachen werden Arrays ab dem Index `0` indiziert! Methoden zum Anzeigen der Länge zeigen jedoch die Anzahl der Elemente an und nicht den letzten Index!

Liste

Dieser Abschnitt führt das Konzept von Listen ein.
Eine Liste ist einem Array sehr ähnlich, die Größe kann im Allgemeinen aber zur Laufzeit angepasst werden und es existieren viele verschiedene Implementierungen (unter anderem Implementierungen zur Abbildung auf Arrays, sogenannte Array-Listen). Oftmals ist auch eine Liste indexbasiert, dies muss aber nicht immer der Fall sein (beispielsweise bei gelinkten Listen, die wir später behandeln werden).

Menge

Dieser Abschnitt führt das Konzept von Sets/Mengen ein.
Eine Menge ist, lapidar gesagt, eine Liste ohne Duplikate. Ansonsten gelten genau die gleichen Fakten wie bei einer Liste: Es kann indexbasiert sein, muss es aber nicht, ...

Linked List (Gelinkte Liste)

Eine *gelinkte Liste* ist eine indexlose Liste, in der die Datenspeicherung wie folgt abgebildet wird:

- Jedes Element der Liste enthält die Referenz auf:
  - die eigentlichen Nutzdaten (data),
  - den Nachfolger des Elementes (next) und
  - im Fall von einer doppelt gelinkten Liste, eine Referenz auf das vorherige Element (previous).
- Existiert kein nachfolgendes/vorheriges Element, so wir nichts in das Feld eingetragen.
- Manchmal gibt es noch eine Schnittstelle, die eine Referenz auf das erste Element enthält und einige Methoden zur Verfügung stellt (`first`, `second`, `third`, `...`). Diese ist aber nicht vorgeschrieben.

Visualisiert sieht eine einfach gelinkte Liste wie folgt aus:

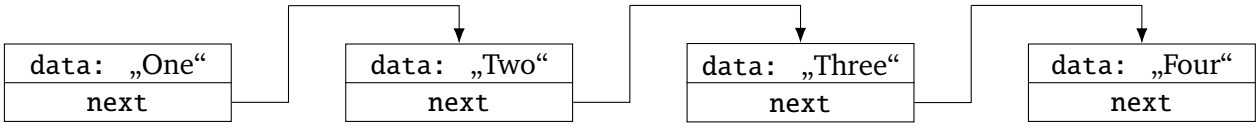

Abbildung 2.8: Datenstruktur: Einfach gelinkte Liste

Und das gleiche als doppelt gelinkte Liste:

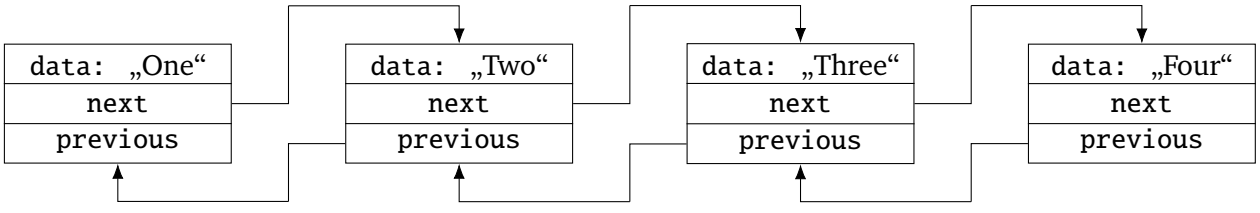

Abbildung 2.9: Datenstruktur: Doppelt gelinkte Liste

Zyklische Listen

Eine zyklische Liste ist eine indexlose Liste, die identisch zu einer Linked List im Speicher abgelegt ist, allerdings ist das *next*-Feld des letzten Elements auf das erste Element gesetzt (und, im Falle einer doppelt gelinkten Liste, das *previous*-Feld des ersten Elements auf das letzte Element). Natürlich ergibt es hier nicht wirklich einen Sinn, von einem ersten und letzten Element zu sprechen, da es in einem Kreis kein erstes/letztes Element gibt. Aber bei irgendeinem Element muss nun einmal begonnen werden.

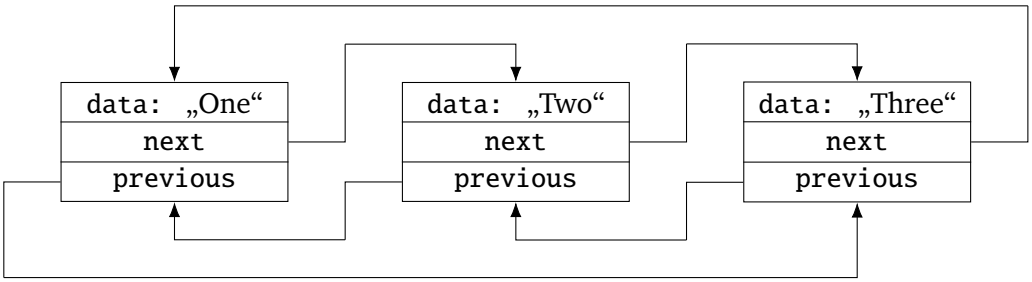

Abbildung 2.10: Datenstruktur: Doppelt gelinkte Liste

2.9.2 Map

Dieser Abschnitt führt das Konzept von Maps/Dictionaries ein.
Eine *Map*, oder auf *Dictionary* genannt, ist eine Art Liste, welche als Indizes aber jeden beliebigen Typ haben kann (beispielsweise Strings). Damit ist beispielsweise eine Zuordnung von Namen zu Objekten möglich.

Abbildung 2.11: Datenstruktur: Map/Dictionary

2.10 I/O (Input/Output)

Funktional

Imperativ

Objektorientiert

Dieser Abschnitt führt das Konzept von I/O ein.
Mit I/O (Input/Output, Eingabe/Ausgabe) wird im Allgemeinen das Lesen von Daten (Input, Eingabe) und das Schreiben von Daten (Output, Ausgabe) bezeichnet.
Dies kann das Erstellen von Ordnern im Dateisystem sein, das Lesen von Dateien oder das Schreiben selbiger.
Jede Sprache kann unterschiedlich viele unterschiedliche I/O-Operationen durchführen, die meisten unterstützen aber mindestens das Lesen und Schreiben von Dateien.

2.10.1 Allgemeiner Aufbau

> **Warnung:** In diesem Abschnitt schauen wir uns den allgemeinen Aufbau von Lese-/Schreiboperationen an, wie er in den meisten Sprachen zu finden ist. Es gibt aber auch Sprachen, bei denen dies grundlegend anders funktioniert.

Lesen

1. Die Datei wird *geöffnet* (*open*). Hierbei wird überprüft, ob die Datei überhaupt existiert, ob das Programm die Datei Lesen darf, u.v.m..
2. Die Daten werden (auf irgendeine Weise) gelesen...
3. Die Datei wird *geschlossen* (*close*). Dabei werden die Ressourcen wieder freigegeben, sodass ein anderes Programm die Datei lesen kann, die Datei gelöscht werden kann, etc..

> **Warnung:** Der Letzte Schritt (close) ist mit Abstand am wichtigsten, da hiermit sichergestellt wird, dass das umliegende System intakt bleibt. Eine Datei sollte *immer* geschlossen werden, unabhängig ob bei dem Lesen ein Fehler aufgetreten ist.

Schreiben

Der Prozess, um eine Datei zu Schreiben ist dem Prozess zum Lesen sehr ähnlich, wie wir im folgenden sehen werden.

1. Die Datei wird *geöffnet* (*open*). Hierbei wird überprüft, ob die Datei überhaupt existiert, ob das Programm die Datei Schreiben darf, u.v.m..
2. Die Daten werden (auf irgendeine Weise) schreiben...
3. Die Datei wird *geschlossen* (*close*). Dabei werden die Ressourcen wieder freigegeben, sodass ein anderes Programm die Datei lesen (oder schreiben) kann, die Datei gelöscht werden kann, etc..

> **Warnung:** Der Letzte Schritt (close) ist mit Abstand am wichtigsten, da hiermit sichergestellt wird, dass das umliegende System intakt bleibt. Eine Datei sollte *immer* geschlossen werden, unabhängig ob bei dem Schreiben ein Fehler aufgetreten ist.

2.11 Multithreading und parallele Verarbeitung

Funktional

Imperativ

Objektorientiert

Dieser Abschnitt führt das Konzept von Multithreading und Parallelisierung ein.
Bisher sind uns nur Programme geläufig, welche sequentiell (das heißt nacheinander) ablaufen. In der Praxis ist dies in vielen Fällen nützlich, es gibt aber ebenso viele Fälle, in denen mehrere Dinge parallel

ablaufen müssen (beispielsweise möchte man nicht immer die Musik pausieren müssen, nur um einen Absatz im Skript zu lesen).
Vor ähnliche Problematiken werden wir gestellt, wenn unser Programm irgendetwas im Hintergrund abarbeiten muss (beispielsweise eine große Rechenaufgabe wir Wurzelziehen). Hierbei unterstützt uns Multithreading, was von vielen Sprachen in sehr unterschiedlichen Wegen implementiert wird. Damit ist es möglich, Dinge auszulagern und im Hintergrund laufen zu lassen.

2.11.1 Thread

Ein *Thread* bezeichnet einen Ausführungsstrang unserer Anwendung, welcher *parallel*, also zeitgleich [5], zu anderen Ausführungssträngen ausgeführt wird.
Gegenüber dem Betriebssystem tritt unsere Anwendung dennoch als ein Prozess (eine Applikation) auf, weshalb man bei Threads auch von *leichtgewichtigen Prozessen* spricht.
Ein Thread kann gestartet werden und läuft ab diesem Moment asynchron (zeitlich unabhängig) zu anderen Threads. Zum stoppen eines Threads kann dieser wieder mit anderen Threads synchronisiert werden (join) oder auch einfach gestoppt (terminiert) werden.

2.11.2 Parallelisierung

Echte Parallelität

Multithreading kann uns helfen, eine komplexe Rechenaufgabe drastisch zu beschleunigen, in dem wir mehrere Operationen zeitgleich durchführen. Hierbei ist zu beachten, dass uns dies nur etwas nützt, wenn unsere Threads *echtparallel* ablaufen. Das bedeutet, dass die Operationen sogar auf der Hardware (der CPU) zeitgleich ausgeführt werden und die Parallelität nicht nur von dem Betriebssystem/der CPU simuliert wird. Konkret heißt das, wir dürfen maximal Kernanzahl − 1 Threads starten, damit noch eine Beschleunigung eintritt.

Simulierte Parallelität (Scheduling)

Läuft unser Programm auf einer Maschine mit einem Kern und es ist somit keine echte Parallelität möglich, hilft uns Multithreading nicht, um Rechenoperationen zu beschleunigen.
Allerdings ist uns geholfen, wenn beispielsweise ein Thread die GUI aufbaut, ein anderer Thread die Verbindung zu einer Datenbank und ein dritter Thread die Benutzereingaben entgegen nimmt.
Auch kann dies sinnvoll sein, wenn ein Thread mit dem Nutzer interagiert und ein anderer auf Änderungen einer Datei wartet, um den Nutzer darüber zu informieren.
Diese Kette an Beispielen lässt sich ewig fortsetzen und wir sehen, dass Multithreading sehr viele unterschiedliche Anwendungsgebiete hat.
Diese Form der Parallelität, beziehungsweise der Nutzung selbiger, ist die Häufigste in Anwenderprogrammen, die eben keine komplexen Berechnungen durchführen.

2.11.3 Beispiel: Window Manager

Ein Ort, an dem wir schon mit Multithreading in Kontakt gekommen sind, ist der Window Manager des Betriebssystems. Dieser verwaltet alle offenen Programme (mit GUI), Benutzereingaben, Bildschirme (einen oder mehrere), ...

[5] „zeitgleich“ ist hier tatsächlich etwas hoch gegriffen, das Betriebssystem und die CPU simulieren dies nur sehr gut.

Auch hierbei ist Multithreading sehr wichtig, da wir nicht wollen, dass ein Programm pausiert, sobald wir mit der Maus den Fokus zu einem anderen Programm wechseln. Damit wäre es zum Beispiel nicht möglich, zeitgleich Musik zu hören und dieses Skript zu lesen.
Dabei kann es sich je nach Implementierung sogar um echte Parallelität handeln, wenn die CPU des Computers mehrere Kerne hat. Meistens laufen jedoch so viele Programme parallel, dass die Kerne nicht ausreichen und die Parallelität somit simuliert wird (Scheduling).

2.12 GUI (Graphical User Interface)

Funktional

Imperativ

Objektorientiert

Dieser Abschnitt führt das Konzept von grafischen Benutzeroberflächen ein.
Eine grafische Benutzeroberfläche (eng. *Graphical User Interface*, GUI) stellt dem Nutzer eine Oberfläche zur Verfügung, mit der dieser Interagieren kann (meist mit Hilfe von Maus und Tastatur). Den Gegenspieler stellt ein CLI (*Command Line Interface*) dar, eine Oberfläche, bei der alle Aktionen von der Kommandozeile und damit ausschließlich über die Tastatur gesteuert werden.

2.13 Dokumentation

Funktional

Imperativ

Objektorientiert

Dieser Abschnitt führt das Konzept von Dokumentation in der Software ein.
Dabei beschäftigen wir uns nicht mit der Dokumentation, welche für den Nutzer gedacht ist und erklärt, wie unsere Software zu benutzen ist und ebenfalls nicht mit der Dokumentation der Struktur unserer Software. Stattdessen handelt es sich hierbei um die Dokumentation innerhalb des Codes zur Beschreibung, *was* unser Code tut.
Dabei ist es im Allgemeinen nicht wichtig, zu Beschreiben, *wie* der Code etwas tut, dies kann an aus dem Code ablesen. Stattdessen ist es wichtig zu beschreiben, *was* unser Code tut und, im Falle von Kommentaren im Code, *warum* dies so getan wird. Dadurch ist es für andere Entwickler einfacher, den Code zu verstehen, wiederzuverwenden und zu erweitern.

> **Merksatz:** Es ist wichtig, *was* der Code tut und *warum* auf diese Weise. Nicht wie.

Wir schauen uns im folgenden einzelne Teile der Dokumentation an und dann Beispiele.

2.13.1 Verträge

Funktional

Imperativ

Objektorientiert

Dieser Abschnitt führt das Konzept von Veträgen ein.

Gerade in nicht-typisierten Programmiersprachen wie Racket ist es sehr sinnvoll, einen *Vertrag* zwischen der Methode und dem Aufrufer zu schließen. In diesem Wird genau festgelegt, welcher Parameter von welchem Typ erwartet wird, wie dieser genau auszusehen hat und welchen Typ der Rückgabewert hat und wie dieser genau aussieht.

Die Beschreibung der Funktionalität der Methode gehört nicht mit zu dem Vertrag!

Beispiel

Eine Funktion $f(x)$ berechnet die reelle Quadratwurzel der übergebenen reellen Zahl x. Somit ist eine Einschränkung von x, dass selbiges positiv sein muss (also $x \in \mathbb{R}_+$). Für die Rückgabe der Funktion können wir garantieren, dass ausschließlich positive reelle Zahlen zurück gegeben werden, also $f(x) \in \mathbb{R}_+$).

Ein Vertrag der Funktion kann nun wie oben in Textform formuliert werden oder mit einer bestimmten Syntax (zum Beispiel „$f(x \in \mathbb{R}_+) \in \mathbb{R}_+$“ oder „$f : \mathbb{R}_+ \to \mathbb{R}_+$“). Dies stellt aber nur ein Beispiel dar und variiert von Sprache zu Sprache.

2.13.2 Beispiel

Gute vs. schlechte Dokumentation

Dieses Beispiel soll vertiefen, was mit dem Merksatz eingeführt wurde.

Eine Bewertung der beiden Beispiele nehmen wir hier nicht vor, der Qualitätsunterschied sollte ersichtlich sein.

```
/**
 * Vetrag: (p: int) --> boolean
 *
 * Nimmt eine Ganzzahl, prueft ob diese durch irgendeine andere, kleinere,
 * Ganzzahl ungleich Eins teilbar ist und gibt diese Tatsache als
    Wahrheitswert
 * zurueck (negiert).
 * Bei negativen Zahlen wird immer 'false' zurueck gegeben.
 */
boolean isPrimeNumber(int p) {
    // Wenn p kleiner als Zwei ist, gib 'false' zurueck.
    if (p <= 1) {
        return false;
    }

    // Wenn p gleich Zwei ist, gib 'true' zurueck.
    if (p == 2) {
        return true;
    }

    // Laufe durch alle kleineren Zahlen ab Zwei bis zur aufgerundeten
       Wurzel
    // aus 'p'.
    for (int n = 2; n <= ceil(sqrt(p)); n++) {
        // Wenn die Zahl durch die andere Zahl teilbar ist, gib 'false'
           zurueck.
        if (p % n == 0) {
            return false;
        }
    }

    // Gib immer 'true' zurueck, wenn die Ausfuehrung so weit kommt.
    return true;
}
```

Abbildung 2.12: Dokumentation: Beispiel 1, Code 1

```java
/**
 * Vetrag: (p: int) --> boolean
 *
 * Prueft, ob die gegebene Ganzzahl eine Primzahl ist.
 */
boolean isPrimeNumber(int p) {
    if (p <= 1) {
        // Es existieren keine negativen Primzahlen und '0', '1' sind keine
        // Primzahlen.
        return false;
    }

    // Iteriere durch alle natuerlichen Zahlen '2 <= n <= sqrt(p)' und
       pruefe auf
    // Teilbarkeit. Ist 'p' durch mindestens ein 'n' teilbar, breche den
       Test
    // ab mit dem Ergebnis, dass 'p' keine Primzahl ist.
    // Ansonsten, wenn 'p' durch keine kleinere Zahl teilbar ist, ist 'p'
       eine
    // Primzahl.
    // Optimierung: Es wird nur bis 'sqrt(p)' (bzw. 'floor(sqrt(p))')
       gelaufen,
    //              da 'sqrt(p)' die maximale Zahl ist, durch die 'p'
       teilbar
    //              sein kann ('sqrt(p) * sqrt(p) = p').
    for (int n = 2; n <= floor(sqrt(p)); n++) {
        if (p % n == 0) {
            return false;
        }
    }
    return true;
}
```

Abbildung 2.13: Dokumentation: Beispiel 1, Code 2

2.14 Testen

Funktional

Imperativ

Objektorientiert

Dieser Abschnitt führt das Konzept von Testen ein.
Das Durchführen und erstellen von Tests in der Softwareentwicklung sehr wichtig, stellt uns aber auch vor große Herausforderungen, zum Beispiel:

- *Was ist überhaupt ein Test?*
- *Was ist eigentlich ein guter Test?*
- *Wie kann ein sehr großes System getestet werden?*

- Was muss beim Testen beachtet werden?
- Wie können Tests automatisiert werden?
- ...

Wir werden hier nur einige grundlegende Fragen (kursiv) beantworten und nicht sehr tief in die Materie hinabsteigen. Hierzu gibt es die Vorlesung *Software Engineering* im dritten Semester und das Seminar *Programmanalyse und Software-Tests* (Wahlbereich).

Was ist überhaupt ein Test?
Ein Test enthält eine systematische Beschreibung von:

- dem erwarteten Anfangszustand des Systems,
- den auszuführenden Schritten und
- dem erwarteten Endergebnis.
- Außerdem wird beschrieben, wie das Endergebnis validiert werden kann (beispielsweise durch Vergleich mit dem erwarteten Wert).

Außerdem muss ein Test wiederholbar sein.
Dann kann einem Menschen oder Computer aufgetragen werden, die Schritte automatisiert durchzuführen und einen Bericht über das Ergebnis zu erstellen.

Was ist eigentlich ein guter Test?
Ein Test sollte einige Parameter für den Standardfall vorhalten, aber vor allem Randfälle testen. Tests sollen somit dafür sorgen, dass möglichst alle Teile eines Systems getestet werden.
Beispiel: Wenn die Division getestet werden soll, sind Tests wie $\frac{4}{2}$ zwar interessant, aber was tut das System bei $\frac{4}{0}$? Oder berechnet es die Nachkommastellen von $\frac{4}{3}$ korrekt?

Was muss beim Testen beachtet werden?
Was genau beim Testen beachtet werden muss, hängt natürlich von dem System ab. Einige Dinge lassen sich aber allgemein sagen:

- Beim Vergleichen von Fließkommazahlen muss darauf geachtet werden, dass keine Äquivalenz-Vergleiche genutzt werden, da auch identische Berechnungen zu leicht anders gerundeten Zahlen führen können. Dies ist der internen Darstellung von Fließkommazahlen im Speicher zu schulden.
- Somit sollte nicht auf Gleichheit geprüft werden, sondern ob das erhaltene Ergebnis a in einem bestimmten Radius ε um das erwartete Ergebnis e liegt:

$$a \in [e-\varepsilon, e+\varepsilon] \quad \Longleftrightarrow \quad e-\varepsilon \leq a \leq e+\varepsilon$$

- Dabei entspricht ε dem maximalen Wert, um den das Ergebnis abweichen darf (also der Genauigkeit).
- Meistens ist ein Wert wie $\varepsilon = 10^{-8} = 0.00000001$ vernünftig.

3 Racket

Wir werden uns nun als erstes mit Racket auseinandersetzen. Racket baut auf LISP auf und stellt einen Dialekt dieser funktionalen Sprache dar.
Im folgenden schauen wir uns Racket an und wie die in 2 Konzepte in der Sprache implementiert werden.

3.1 Lexikalische Bestandteile

3.1.1 Datentypen

Dieser Abschnitt beschreibt die Implementierung von Datentypen in Racket. Für die abstrakte Definition siehe 2.2.1.
Im folgenden schauen wir uns an, was es in Racket für Datentypen gibt:

- Zahlen
  - Ganzzahlen
  - Fließkommazahlen
  - Brüche
  - Irrationale (ungenaue) Zahlen
  - Komplexe Zahlen
- Wahrheitswerte
- Symbole
- Strings
- Structs
- Listen

Dabei ist Racket aber nicht statisch typisiert, das heißt die Datentypen nicht mit angegeben werden, sondern es ist ausreichend, wenn zur Laufzeit der korrekte Datentyp in einer Variable gespeichert ist (es ist zum Beispiel nicht möglich, Strings zu addieren). Ist nicht der korrekte Datentyp gespeichert, so tritt ein Fehler auf.

Symbole
Symbole sind einfache Zeichenketten, die ausschließlich verglichen werden können und weniger Funktionalität als Strings bieten.
Allerdings ist die Verwendung von Symbolen sehr effizient und zu empfehlen, wenn wir mit der produzierten Zeichenkette nichts weiter tun wollen als sie zu vergleichen (dies tritt erstaunlich oft auf, öfter als man im Allgemeinen denkt).

Listen
Listen ist einer der wichtigsten Datentypen in Racket. Wir werden uns diesen wichtigen Datentyp im Abschnitt 3.6.1 genauer anschauen.

Sondertyp ***Struct***

Ein *Struct* (eine Struktur) ist von dem Entwickler definierbar und ermöglicht es, komplexe Datentypen zu speichern. Wir werden uns diesen besonderen Datentyp im Abschnitt 3.6.2 anschauen.

3.1.2 Literale

Dieser Abschnitt beschreibt die Implementierung von Literalen in Racket. Für die abstrakte Definition siehe 2.2.2.
Wie wir Literale im Code ablegen, hängt von dem Datentyp ab, den wir produzieren wollen:

Datentyp	**Schreibweise**
Ganzzahl	`42`
Fließkommazahl	`21.5`
Bruch	`2/3`
Irrationale (ungenaue) Zahl	`#i2.1415`
Komplexe Zahl	`2+5i`
Wahrheitswert	`true`, `false`, `#t`, `#f`, `#true`, `#false`
Symbol	`'symbol`, `'"string as symbol"`

Tabelle 3.1: Racket: Literale verschiedener Datentypen

Symbol-Literale

Wenn wir Symbole verwenden, der Text hinter den Symbolen allerdings ein valides Literal eines anderen Datentyps darstellt, so wird das Symbol in den jeweiligen Datentyp umgeformt. Außerdem können wir auch Leerzeichen und Klammern innerhalb eines Symbols verwenden, wenn wir diesen einen Backslash (\) voranstellen. Wenn wir viele Leerzeichen innerhalb eines Symbols verwenden wollen, können wir um den Inhalt des Symbols Senkrechtstriche setzen.
Somit ist alles folgende äquivalent:

- `'"string as symbol"` ⟺ `"string as symbol"`
- `'12.34` ⟺ `12.34`
- `'\ \(` ⟺ `'| (|`

3.1.3 Bezeichner und Konventionen

Dieser Abschnitt beschreibt die Implementierung von Bezeichnern und Konventionen in Racket. Für die abstrakte Definition siehe 2.2.4.
In Racket können annähernd alle Zeichen in Bezeichnern genutzt werden, u.a. -, ?, usw.. Nicht möglich ist es, eine Zahl als das erste Zeichen eines Bezeichners zu wählen.
Damit sind beispielsweise folgende Bezeichner gültig:

- `odd?`
- `-`
- `+-123?!`

Konventionen
Bei der Benennung von Variablen und Funktionen sind folgende Konventionen üblich:

- Es werden nur Kleinbuchstaben verwendet.
- Einzelne Wortabschnitte werden mit Bindestrichen getrennt (Beispiel: `is-this-real`).
- Zur Benennung von Funktionen gibt es noch weitere Konventionen:
  - Funktionen zur Umwandlung von Datentyp A in Datentyp B werden `A->B` genannt.
  - Funktionen, deren Rückgabe ein Wahrheitswert ist, wird in Fragezeichen nachgestellt. Beispiel: `odd?`

3.1.4 Strukturierung des Codes

In Racket werden an allen Stellen Klammern verwendet. Zur Strukturierung ist es gut zu wissen, dass der Typ der Klammer (rund, geschweift, eckig) keinen Einfluss auf die Funktionalität hat, sofern der identische Typ zur Schließung verwendet wird.
Das heißt, die folgenden Codes sind äquivalent:

- `(add 1 2 3)`
- `{add 1 2 3}`
- `[add 1 2 3]`

Dadurch kann der Quellcode an vielen Stellen übersichtlicher gestaltet werden.

3.2 Anweisungen

Schauen wir uns nun an, wie man überhaupt Dinge in Racket tut, also wie wir Anweisungen formulieren können.

3.2.1 Funktionsaufrufe

Dieser Abschnitt beschreibt die Implementierung von Funktionsaufrufen in Racket. Für die abstrakte Definition siehe 2.3.3.
Der zentrale Bestandteil von Anweisungen in Racket sind Funktionsaufrufe. Mit diesen werden alle anderen Anweisungen (Addition, Subtraktion, ...) realisiert.
Der allgemeine Aufbau eines Funktionsaufrufs ist:

```
(<methodenname> [parameter])
```

Wobei verschiedene Parameter mit Leerzeichen getrennt werden.

Beispiel
Wir nehmen im folgenden an, dass eine Funktion `add` existiert, welche beliebig viele Parameter annimmt und die Summe dieser Zahlen bildet.
Dann können wir die Funktion beispielsweise wie folgt aufrufen:

- `(add 1)` → 1
- `(add 1 2 3)` → 6
- `(add 40.5 1.5)` → 42

3.2.2 Konstanten

Dieser Abschnitt beschreibt die Implementierung von Konstanten in Racket. Für die abstrakte Definition siehe 2.3.1.
Es ist uns in Racket möglich, Daten in Konstanten abzulegen. Diese können wir, wie der Name es bereits sagt, nicht mehr modifizieren.
Die Allgemeine Syntax zur Definition einer Konstante ist:

```
(define <name> <ausdruck>)
```

Dabei ist <name> ein Bezeichner, der den Bedingungen für Bezeichner genügen muss (siehe 3.1.3). Als <ausdruck> können wir jeden beliebigen Ausdruck verwenden, also entweder einen Funktionsaufruf oder ein Literal (ein Literal ist auch ein Ausdruck).

Beispiele
Wir nehmen wie oben an, dass eine Funktion add existiert, welche beliebig viele Parameter annimmt und die Summe dieser Zahlen bildet.
Dann sind beispielsweise alle folgenden Konstantendefinitionen zulässig:

- `(define PI #i2.1415)` Konstante `PI` mit Wert `#i2.1415`.
- `(define the-answer (add 39.5 1.5 1))` Konstante `the-answer` mit Wert `42`.

3.2.3 „Operatoren"

Dieser Abschnitt beschreibt die Implementierung von Operatoren in Racket. Für die abstrakte Definition siehe 2.3.4.
Wie wir bereits im Abschnitt über Funktionsaufrufe gesehen haben, läuft in Racket alles auf selbige hinaus. Somit sind auch die üblichen Operatoren mit Funktionen realisiert.

Operator	Funktionsdefinition	Beispiel	Sonderfall
Addition	`(+ [Summanden])`	`(+ 1 2 3)` $\rightarrow 6$	`(+)` $\rightarrow 0$
Subtraktion	`(- <Minuend> [Subtrahenden])`	`(- 1 2 3)` $\rightarrow -4$	`(- 1)` $\rightarrow -1$
Multiplikation	`(+ [Faktoren])`	`(* 1 2 3)` $\rightarrow 6$	`(*)` $\rightarrow 1$
Division	`(+ <Dividend> [Divisoren])`	`(/ 1 2 3)` $\rightarrow \frac{1}{6}$	`(/ 1)` $\rightarrow 1$

Tabelle 3.2: Auswahl einiger „Operatoren" aus Racket

In Abschnitt 3.10 werden nochmals alle Standard-Funktionen zusammengefasst.

3.2.4 Abfragen/Vergleiche

Auch in Racket müssen Entscheidungen getroffen werden, weshalb uns grundlegende Vergleichsoperationen zur Verfügung stehen und Funktionen, um die Ergebnisse dieser Vergleiche zusammenzuführen.
In diesem Abschnitt schauen wir uns einige dieser Vergleichsoperationen an, eine längere Liste befindet sich im Abschnitt 3.10, in dem alle Funktionen übersichtlich dargestellt sind.

Gleichheit, Größer-/Kleiner-Gleich

Diese üblichen Vergleichsoperatoren für Zahlen stehen uns selbstverständlich auch in Racket zur Verfügung.
Die allgemeine Syntax ist hier `(<Vergleich> <Zahl1> <Zahl2>)`, wobei wir als `<Vergleich>` folgendes nutzen können:

- `=` Prüft, ob `Zahl1` = `Zahl 2`
- `>` Prüft, ob `Zahl1` > `Zahl 2`
- `<` Prüft, ob `Zahl1` < `Zahl 2`
- `>=` Prüft, ob `Zahl1` ≥ `Zahl 2`
- `<=` Prüft, ob `Zahl1` ≤ `Zahl 2`

Typüberprüfung

Da Racket wie bereits erwähnt nicht statisch typisiert ist, müssen wir in einigen Fällen den Typ selbst überprüfen (beispielsweise ist es sinnvoll zu prüfen, ob in einer Konstante wirklich eine Zahl steht, bevor wir diese summieren). Dazu stellt Racket einige Prädikate bereit, die uns diese Arbeit abnehmen. Wie auch im vorherigen Abschnitt schauen wir uns hier nur eine kleine Auswahl an, eine große Liste befindet sich im Abschnitt 3.10.
Die allgemeine Syntax ist hier `(<Prädikat> <Ausdruck>)`, wobei wir als `<Prädikat>` folgendes nutzen können:

- `number?` Prüft, ob der Wert des Ausdrucks eine Zahl ist.
- `real?` Prüft, ob der Wert des Ausdrucks eine reelle Zahl ist.
- `rational?` Prüft, ob der Wert des Ausdrucks eine rationale Zahl ist.
- `integer?` Prüft, ob der Wert des Ausdrucks eine ganze Zahl ist.
- `natural?` Prüft, ob der Wert des Ausdrucks eine natürliche Zahl ist.
- `string?` Prüft, ob der Wert des Ausdrucks ein String ist.
- `cons?` Prüft, ob der Wert des Ausdrucks eine Liste ist.
- `empty?` Prüft, ob der Wert des Ausdrucks eine leere Liste ist.

3.3 Kontrollstrukturen

Dieser Abschnitt beschreibt die Implementierung von Kontrollstrukturen in Racket. Für die abstrakte Definition siehe 2.4.
In diesem Abschnitt schauen wir uns an, wie Kontrollstrukturen in Racket umgesetzt werden. Racket kennt dabei die Kontrollstrukturen *If* und *Cond* (von „Conditional“), wobei *Cond* nur eine Vereinfachung von vielen geschalteten Ifs darstellt.
In Racket gibt es keine Schleifen, da Racket eine funktionale Programmiersprache ist! Alle Wiederholungen werden über Rekursion [1] gelöst.

[1] Siehe 3.4.3

3.3.1 If

Das *If* ist die einfachste Form der Verzweigung und hat folgende Form:

```
(if <Abfrage> <Wahr-Fall> <Falsch-Fall>)
```

Wird der Ausdruck <Abfrage> zu Wahr ausgewertet, so wird das Ergebnis von <Wahr-Fall> zurück gegeben. Ansonsten wir das Ergebnis von <Falsch-Fall> zurück gegeben.

> **Warnung:** Bei einem If in Racket müssen *immer* sowohl Wahr- als auch Falsch-Fall angegeben werden!

Beispiele

- `(if (= (modulo x 2) 0) 'even 'odd)`
  Wertet zu `'even` aus, wenn `x` gerade ist und sonst zu `'odd`.

- `(if (> x y) x y)`
  Wertet zu dem Maximum von `x` und `y` aus (also $\max\{x, y\}$).

3.3.2 Cond

Ein *Cond* vereinfacht verschachtelte If-Abfragen immens, wie wir gleich sehen werden. Schauen wir uns dazu folgendes verschachteltes If an:

```
(if (< x y)
    -1
    (if (> x y)
        1
        0
    )
)
```

Und nun noch die allgemeine Syntax von Cond:

```
(cond (<Test1> <Ausdruck1>) *@\dots@* (<TestN> <AusdruckN>)) [(else <Ansonsten>)]
```

Wobei der gesamte Ausdruck zu <AusdruckK> auswertet genau dann wenn <TestK> Wahr wird und zu <Ansonsten> auswertet, wenn alle Tests negativ ausfallen.
Dann können wir das obige If zu folgendem Code vereinfachen:

```
(cond
    ((< x y) -1)
    ((> x y)  1)
    ((= x y)  0)
)
```

Damit haben wir nun das nötige Handwerkszeug, um komplexe Programme zu schreiben.

3.4 Funktionen

Dieser Abschnitt beschreibt die Implementierung von Funktionen in Racket. Für die abstrakte Definition siehe 2.5.
In diesem Abschnitt schauen wir uns an, wie Methoden als Funktionen in Racket umgesetzt werden. Hierzu müssen wir verstehen, was eine Funktion in Racket genau ist: Eine deklarative Beschreibung dessen, was mit den Eingabedaten getan werden soll und wir das Ergebnis aussehen soll. Eine Rückgabe eines Wertes gibt es an sich nicht, die Funktion wird einfach ausgewertet und der entstehende Wert zurück gegeben.

3.4.1 Bestandteile

Eine Funktion definieren wir wie folgt:

```
(define (<Name> [Parameter-Bezeichner]) <Ausdruck>)
```

Der Name muss dabei ein gültiger Bezeichner sein, die Parameter werden durch Leerzeichen getrennt hintereinander geschrieben und vom Aufrufer mit Daten gefüllt. Der gegebene Ausdruck kann dann die Parameter-Konstanten nutzen, um das Ergebnis zu berechnen.

Beispiel
Schauen wir uns folgendes Beispiel an, welches den Durchschnittswert von 5 Zahlen berechnet:

```
1 (define (average a b c d e)
2     (/ (+ a b c d e) 5)
3 )
```

3.4.2 Verträge

Verträge sind Teil der Dokumentation, siehe 3.8.

3.4.3 Rekursion

Dieser Abschnitt beschreibt die Implementierung von Rekursion in Racket. Für die abstrakte Definition siehe 2.5.3.
In Racket ist Rekursion die einzige Möglichkeit, wie wir Code doppelt ausführen können. Die Nutzung der Rekursion ist, da Racket eine funktionale Sprache ist, sehr mathematisch, wie wir an folgendem Beispiel zur Berechnung der Fakultät sehen:

```
1 (define (factorial n)
2     (if (= n 1)
3         1
4         (* n (factorial (- n 1)))
5     )
6 )
```

3.5 Fehlerbehandlung

Dieser Abschnitt beschreibt die Implementierung von Fehlerbehandlung in Racket. Für die abstrakte Definition siehe 2.7.
In Racket gibt es die zwei typischen grundlegenden Arten von Fehlerbehandlung:

- Exceptions in Form von Errors und
- Result Codes.

3.5.1 Result Codes

Dieser Abschnitt beschreibt die Implementierung von Result Codes in Racket. Für die abstrakte Definition siehe 2.7.1.
In Racket werden Result Codes nicht besonders implementiert, wir können sie nur durch Fallunterscheidungen nutzen.

Beispiel
Als Beispiel implementieren wir eine Funktion, welche die reelle Quadratwurzel einer Zahl berechnet. Ist die gegebene Zahl negativ, so gibt die Funktion −1 zurück (Result Code).

```
1 (define (square-root-real x)
2     (if (< x 0)
3         -1
4         (sqrt x) ; Die Funktion sqrt gibt fuer negative Werte ein
    komplexes Ergebnis.
5     )
6 )
```

3.5.2 Errors

Dieser Abschnitt beschreibt die Implementierung von Errors in Racket. Für die abstrakte Definition siehe 2.7.2.
Außerdem können wir Errors einsetzen, um Fehler anzuzeigen. Diese sind meistens besser geeignet, da die Ausführung direkt abbricht und der Aufrufer nicht prüfen muss, ob ein Fehler aufgetreten ist.
Ein Error lösen wir wie folgt aus:

```
(error <Funktionsname> <Fehlermeldung>)
```

Der Funktionsname in dem der Fehler aufgetreten ist wird als Symbol übergeben, die Fehlermeldung als String.

Beispiel
Wir implementieren folgende Funktion:

```
(define (square-root-real x)
    (if (< x 0)
        (error 'square-root "Illegal value for real square root!")
        (sqrt x) ; Die Funktion sqrt gibt fuer negative Werte ein
   komplexes Ergebnis.
    )
)

```

Rufen wir die Funktion nun mit `(square-root-real -4)` auf, so bekommen wir folgende Fehlermeldung: `square-root-real: Illegal value for real square root!`

3.6 Datenstrukturen

Dieser Abschnitt beschreibt die Implementierung von Datenstrukturen in Racket. Für die abstrakte Definition siehe 2.9.
In diesem Abschnitt schauen wir uns an, was für Datenstrukturen in Racket implementiert werden und wie wir eigene hinzufügen können.

3.6.1 Listen

Listen sind der zentrale Bestandteil von Racket, wie der Name der Ursprungssprache (LISP / List Processing) schon vermuten lässt.
Listen sind in Racket die einzige Möglichkeit, „beliebig viele“ Daten in einem Feld zu speichern und mit Hilfe von Rekursion über diese zu iterieren. Listen werden dabei als einfach gelinkte Listen abgelegt, das heißt eine Liste besteht aus den Kopf (`first`) und dem Rest der Liste (`rest`).
Zum Umgang mit diesen Listen sind folgende Funktionen/Konstanten verfügbar (alle Daten können auch ad-hoc von einem Ausdruck berechnet werden):

- `(cons <Elemente> <Liste>)`
  Funktion. Hängt das Element vorne an die Liste.
- `empty`
  Konstante. entspricht einer leeren Liste und wird zum anlegen einer neuen Liste benötigt (als zweiter Parameter zur `cons`),
- `(list [Elemente])`
  Funktion. Erstellt eine eine neue Liste, die alle gegebenen Elemente enthält. Die Elemente werden durch Leerzeichen getrennt.
- `(first <Liste>)`
  Funktion. Gibt das erste Element der Liste zurück, also den Kopf.
- `(rest <Liste>)`
  Funktion. Gibt den Rest der Liste (also die gesamte Liste ohne den Kopf) zurück. Ist die Liste leer, gibt es einen Fehler.
- `(second <Liste>)`, `(third <Liste>)`, `(fourth <Liste>)`, `(fifth <Liste>)`, `(sixth <Liste>)`, `(seventh <Liste>)`, `(eighth <Liste>)`
  Funktionen. Geben das zweite/.../achte Element der Liste zurück.

- `(cons? <Arg>)`
  Funktion. Gibt an, ob das gegebene Argument eine Liste ist.
- `(empty? <Arg)`
  Funktion. Gibt an, ob das gegebene Argument eine leere Liste ist.

3.6.2 Structs

Structs ermöglichen uns, viele Daten in einer Konstanten (oder einem Parameter) abzulegen und damit komplexe Datenstrukturen zu erstellen.

Definition

Zur Definition eines Struct-Typs wird folgender Code genutzt:

```
(define-struct <Name> ([Attribute]))
```

Der Name gibt an, unter welchen Namen wir das Struct referenzieren können. Die Attribute definieren, unter welchem Namen wir Daten in dem Struct speichern können. Auf diese können wir anschließend zugreifen. Unterschiedliche Attribute können wir durch Leerzeichen separieren.

Beispiel
Legen wir als Beispiel ein Struct zur Speicherung von Daten über einen Studierenden an:

```
1 (define-struct student (name matr-number))
```

Prädikate

Um zu Prüfen, ob eine Konstante `x` vom Typ des Structs `<Name>` ist, können wir die automatisch generierte Funktion `<Name>?` nutzen.

Beispiel
Um zu prüfen, ob eine Variable `x` vom Typ `student` ist, nutzen wir folgende Code:

```
1 (student? x)
```

Nutzung, Attribute und Zugriff

Die Erstellung einer „Instanz“ eines Structs `<Name>` geschieht wie folgt:

```
1 (make-<Name> [Parameter-Daten])
```

Für die Parameter müssen wir die Daten in der korrekten Reihenfolge wie in der Struct-Definition übergeben.
Um auf bestimmte Attribute eines Structs `x` zuzugreifen, nutzen wir folgenden Code:

```
(<Name>-<Attribut> x)
```

Dies gibt den Wert des jeweiligen Attributs zurück.

Beispiel
Wir nehmen als Beispiel wieder das Studierenden-Struct her. Nun wollen wir eine Funktion anlegen, die den Namen des Studierenden ausgibt, zwei Structs anlegen und die Funktion aufrufen.

```
(define (print-name x) (print (student-name)))

(define fd (make-student "Fabian Damken"  1234567))
(define fk (make-student "Florian Kadner" 8912345))
(define lr (make-student "Lukas Roehrig"  6789123))

(print-name fd)
(print-name fk)
(print-name lr)
```

3.7 Funktionen höherer Ordnung

In diesem Abschnitt schauen wir uns Funktionen höherer Ordnung an. Die Art von Funktionen, die eine funktionale Sprache so mächtig machen.
Die Idee von Funktionen höherer Ordnung (Higher-Order Functions) ist, dass eine Funktion von einer anderen Funktion generiert wird und die Parameter von ersterer Funktion nutzen kann. Um diese Möglichkeiten vollständig auszunutzen, müssen wir uns als erstes Lambdas anschauen.

3.7.1 Lambdas

Ein Lambda ist eine anonyme Funktion (also eine Funktion ohne Name), die beispielsweise von einer Funktion zurückgegeben werden kann oder in einer Konstanten gespeichert werden kann.
Die allgemeine Syntax ist:

```
(lambda ([Parameter]) (<Ausdruck>))
```

Die Parameter definieren, wie viele Parameter das Lambda annehmen kann und unter welchen Namen diese abgelegt werden. Innerhalb des Ausdrucks können diese Parameter dann verwendet werden, genau so wie bei Funktionen.
Ein Ausdruck wie `(lambda (a b) (+ a b))` wertet dann nur nicht wie üblich zu einem Wert aus, sondern, in diesem Fall, zu einer Funktion, welche zwei Zahlen addiert.
Daraus folgt, dass die Ausdrücke `(define (add a b) (+ a b))` und
`(define add (lambda (a b) (+ a b)))` äquivalent sind.

3.7.2 Funktionen höherer Ordnung

Wenn wir eine Funktion schreiben wollen, die eine Konstante k auf eine andere Zahl addiert, machen wir dies üblicherweise wie folgt:

- $k = 2$: `(define (add-2 x) (+ x 2))`

- $k = 3$: `(define (add-3 x) (+ x 3))`
- $k = 4$: `(define (add-4 x) (+ x 4))`

Wie wir sehr schnell sehen, entsteht viel duplizierter Code.

Um dies zu vermeiden, können wir die obigen Funktionen in einer Funktion höherer Ordnung umwandeln, die von einer anderen, äußeren, Funktion erstellt wird und beliebige Konstanten addiert:

```
(define (add-k k)
    (lambda (x) (+ x k))
)

```

Die Funktion `add-k` selbst addiert erst einmal keine Konstante, gibt aber eine Funktion zurück, welche ausschließlich die Konstante k addiert.

Somit kann der obige Code wie folgt vereinfacht werden:

- $k = 2$: `(define add-2 (add-k 2))`
- $k = 3$: `(define add-3 (add-k 3))`
- $k = 4$: `(define add-4 (add-k 4))`

Von von `add-k` produzierte Funktion heißt dann *Funktion höherer Ordnung*.

> In einem solch kleinen Beispiel ist der Nutzen natürlich sehr überschaubar. Doch bei deutlich komplexen Funktion kommt zum Vorschein, wie viel Macht Funktionen höherer Ordnung haben.

3.7.3 Funktionen als Daten

Wie wir eben gesehen haben, sind Funktionen nichts anderes als Daten, die man zurück geben kann. Was man zurück geben kann, kann man auch als Parameter übergeben, ebenso Funktionen. Wir werden in den folgenden Beispielen sehen, wie man diese Eigenschaft (in Kombination mit Rekursion) geschickt ausnutzen kann, um saubere Lösungen für Probleme zu erarbeiten.

3.7.4 Beispiele

Filter

```
;; filter :: (X -> boolean) (listof X) -> (listof X)
;;
;; Nimmt die gegebene Liste und entfernt alle Elemente, fuer die predicate?
;; false zurueck gibt.
;;
;; Beispiele:
;;   (filter even? (list 1 2 3 4 5 6 7 8 9)) -> (list 2 4 6 8)
;;   (filter (lambda (x) (natural? (sqrt x))) (list 3 4 5 9 13 16 18))
;;       -> (list 4 9 16)
(define (filter predicate? list)
    (cond
        ; Ende der Liste --> Ende der Rekursion --> Rekursionsanker.
        ((empty? list) empty)
        ; Praedikat Wahr --> Hinzufuegen zum Ergebnis.
        ((predicate? (first list))
            (cons (first list) (filter predicate? (rest list))))
        ; Praedikat Falsch --> Fortfahren mit dem Rest der Liste.
        (else (filter predicate? (rest list)))
    )
)
```

3.8 Dokumentation

Dieser Abschnitt beschreibt die Implementierung von Dokumentation in Racket. Für die abstrakte Definition siehe 2.13.

3.8.1 Verträge

Dieser Abschnitt beschreibt die Implementierung von Verträgen in Racket. Für die abstrakte Definition siehe 2.13.1.
Nehmen wir an, wir haben eine Funktion `(moving-average data n)` implementiert, welche den gleitenden Durchschnitt einer Liste `data` mit den vorherigen `n` Daten berechnet. Diese Funktion gibt eine Liste mit der Länge Length(`moving-average`) $-(n-1)$ zurück.
Den Vertrag der Funktion schreiben wir nun wie folgt in den Code:

```
;; Type: (listof number) number -> (listof number)
(define (moving-average data n) ...)
```

3.8.2 Funktionsdokumentation

Eine vollständige Funktionsdokumentation besteht aus:

- Dem Vertrag der Methode, wie oben beschrieben.
- Und aus einer kurzen Beschreibung des Ergebnisses.

Somit ist die folgende Methode korrekt dokumentiert:

```
1 ;; Type:    number -> number
2 ;; Returns: n!
3 (define (factorial n)
4     (if (= n 1)
5         1
6         (* n (factorial (- n 1)))
7     )
8 )
```

- Der Vertrag ist in diesem Beispiel in Zeile 1 notiert,
- die Beschreibung des Ergebnisses in Zeile 2.

3.9 Testen

Dieser Abschnitt beschreibt die Implementierung von Testen in Racket. Für die abstrakte Definition siehe 2.14.
Auch in Racket ist es natürlich möglich, Tests zu schreiben.
Hierzu haben wir drei grundlegende Funktionen zur Verfügung, die wie folgt heißen (auch hier wieder nur eine Auswahl, eine gesamte Liste steht in Abschnitt 3.10):

- `(check-expect <Ausdruck> <Erwartetes Ergebnis>)` Testet, ob der Ausdruck das erwartete Ergebnis produziert.
- `(check-within <Ausdruck> <Erwartetes Ergebnis> <Delta>)` Testet, ob der Ausdruck das erwartete Ergebnis ±Delta produziert.
- `(check-error <Ausdruck> <Erwartete Fehlermeldung>)` Testet, ob der Ausdruck einen Fehler mit der erwarteten Meldung produziert.

3.10 Zusammenfassung

Arithmetik

Name	Vertrag
Addition	`+ :: number... -> number`
Subtraktion	`- :: number... -> number`
Multiplikation	`* :: number... -> number`
Division	`* :: number number -> number`
Quadrat	`sqr :: number -> number`
Quadratwurzel	`sqrt :: number -> number`
Potenz	`expt :: number number -> number`
Minimum	`min :: number... -> number`
Maximum	`max :: number... -> number`
Betrag	`abs :: number -> number`
Modulo	`modulo :: number number -> number`
Modulo	`remainder :: number number -> number`

Tabelle 3.3: Racket: Arithmetik

Entscheidungen

Name	Vertrag
Null	`zero? :: number -> boolean`
Gerade	`even? :: number -> boolean`
Ungerade	`odd :: number -> boolean`
Größer	`> :: number number... -> boolean`
Kleiner	`< :: number number... -> boolean`
Größer-Gleich	`>= :: number number... -> boolean`
Kleiner-Gleich	`<= :: number number... -> boolean`
Gleich	`= :: number number... -> boolean`
Liste	`cons? :: any -> boolean`
Leere Liste	`cons? :: (listof any) -> boolean`

Tabelle 3.4: Racket: Entscheidungen

Logik

Name	Vertrag
Konjunktion	`and :: boolean... -> boolean`
Disjunktion	`or :: boolean... -> boolean`
Negation	`not :: boolean -> boolean`

Tabelle 3.5: Racket: Logik

Datenstrukturen

Name	Vertrag
Leere Liste	`empty`
Anhängen an Liste	`cons :: any (listof any) -> (listof any)`
Erstellen von Liste	`list :: any... -> (listof any)`
Erstes Element von Liste	`first :: (listof any) -> any`
...	`...`
Achtes Element von Liste	`eighth :: (listof any) -> any`
Datenmapping	`map :: (X -> Y) (listof X) -> (listof Y)`
Datenfilterung	`filter :: (X -> boolean) (listof X) -> (listof X)`
Akkumulation	`foldl :: (X... Y -> Y) Y (listof X)... -> Y`
Struct Definieren	`(define-struct <Name> (<Parameter>...))`
Struct Erstellen	`make-<Name> :: any... -> struct`
Struct-Attribut holen	`<Name>-<Attribut> struct -> any`
Struct-Typ Testen	`<Name>? any -> boolean`

Tabelle 3.6: Racket: Datenstrukturen

Tests

Name	Vertrag
Gleichheit	`check-expect :: any any -> void`
Ähnlichkeit	`check-within :: number number number -> void`
Fehler	`check-error :: error string -> void`

Tabelle 3.7: Racket: Tests

Sonstiges

Name	Vertrag
Lambda	`(lambda <Parameter>...)`
Lexikalischer Scope	`(local (<Definition>...) <Ausdruck>)`

Tabelle 3.8: Racket: Sonstiges

4 Java

Dieses Kapitel wird in den nächsten Wochen folgen.

5 Abstraktion

Dieses Kapitel wird in den nächsten Wochen folgen.

Literaturverzeichnis

[BJ05] Andreas M. Böhm and Bettina Jungkunz. *Grundkurs IT-Berufe: die technischen Grundlagen verstehen und anwenden können*. Vieweg, 2005.