TECHNISCHE UNIVERSITÄT DARMSTADT

Kapitel 04a:
Funktionales Programmieren: Grundlagen

Karsten Weihe

TECHNISCHE UNIVERSITÄT DARMSTADT

Rückschau: Objektorientierte Abstraktion

Offenbar geht es in der Informatik immer um Abstraktion. Es gibt aber grundsätzlich verschiedene Modelle für Abstraktion. In der Informatik spricht man von Paradigmen, genauer Programmierparadigmen. Jede Programmiersprache ist durch ein oder mehrere Programmierparadigmen geprägt. Java ist vorrangig durch das objektorientierte Paradigma geprägt, daher wird Java häufig als objektorientierte Sprache bezeichnet.

In diesem Kapitel werden wir ein weiteres Paradigma kennen lernen, das funktionale. Aber zuerst halten wir Rückschau auf das objektorientierte Paradigma.

Objektorientierte Abstraktion

TECHNISCHE UNIVERSITÄT DARMSTADT

- Objekte / Klassen sind die zentralen Bausteine.
  - Jede Subroutine gehört zu einer Klasse.
  - Heißt typischerweise Methode statt Subroutine / Prozedur / Funktion.
- Ein Objekt hat einen momentanen Zustand.
  - Ändert sich durch Schreibzugriffe auf seine Attribute und Aufrufe seiner Methoden.
  - Zeitliche Abläufe sind zu durchdenken.

Eigentlich reicht dafür eine kleine Übersicht wie auf dieser und der nächsten Folie, die bisherigen Fallbeispiele mit Fopbot und geometrischen Figuren füllen diese Übersicht mit Leben.

Objektorientierte Abstraktion

TECHNISCHE UNIVERSITÄT DARMSTADT

- **Objekte / Klassen sind die zentralen Bausteine.**
  - **Jede Subroutine gehört zu einer Klasse.**
  - **Heißt typischerweise Methode statt Subroutine / Prozedur / Funktion.**
- **Ein Objekt hat einen momentanen Zustand.**
  - **Ändert sich durch Schreibzugriffe auf seine Attribute und Aufrufe seiner Methoden.**
  - **Zeitliche Abläufe sind zu durchdenken.**

Das haben wir sowohl bei Fopbot als auch bei geometrischen Figuren gesehen: Zentral sind die Objekte beziehungsweise ihre Typen, die Klassen.

Objektorientierte Abstraktion

TECHNISCHE UNIVERSITÄT DARMSTADT

- Objekte / Klassen sind die zentralen Bausteine.
  - Jede Subroutine gehört zu einer Klasse.
  - Heißt typischerweise Methode statt Subroutine / Prozedur / Funktion.
- Ein Objekt hat einen momentanen Zustand.
  - Ändert sich durch Schreibzugriffe auf seine Attribute und Aufrufe seiner Methoden.
  - Zeitliche Abläufe sind zu durchdenken.

Aus der Mathematik und vielleicht auch aus manchen anderen Programmiersprachen sind Sie gewohnt, Funktionen als unabhängige Entitäten aufzufassen. In vielen Programmiersprachen ist das auch so ähnlich, und so werden wir das in diesem Kapitel auch sehen. Bei Java haben wir eine andere Sicht gesehen: Methodenimplementationen sind Bestandteile einer Klasse; Klassenmethoden werden mit Klasse oder Objekt aufgerufen; Objektmethoden werden mit Objekt aufgerufen.

Objektorientierte Abstraktion

TECHNISCHE UNIVERSITÄT DARMSTADT

- Objekte / Klassen sind die zentralen Bausteine.
  - Jede Subroutine gehört zu einer Klasse.
  - Heißt typischerweise Methode statt Subroutine / Prozedur / Funktion.
- Ein Objekt hat einen momentanen Zustand.
  - Ändert sich durch Schreibzugriffe auf seine Attribute und Aufrufe seiner Methoden.
  - Zeitliche Abläufe sind zu durchdenken.

Der momentane Zustand eines Roboter-Objektes in Fopbot ist – soweit wir es gesehen haben – beschrieben durch die momentane Zeile und Spalte, die Richtung und die Anzahl Münzen. Diese bilden zusammen den Zustand des Roboter-Objektes.

Bei einem Kreisobjekt im geometrischen Beispiel wird der momentane Zustand hingegen beschrieben durch die Koordinaten des Referenzpunktes und durch den Radius.

In Java wird der Zustand eines Objektes also generell durch die Werte in den Attributen seiner Klasse beschrieben.

Objektorientierte Abstraktion

TECHNISCHE UNIVERSITÄT DARMSTADT

- Objekte / Klassen sind die zentralen Bausteine.
  - Jede Subroutine gehört zu einer Klasse.
  - Heißt typischerweise Methode statt Subroutine / Prozedur / Funktion.
- Ein Objekt hat einen momentanen Zustand.
  - Ändert sich durch Schreibzugriffe auf seine Attribute und Aufrufe seiner Methoden.
  - Zeitliche Abläufe sind zu durchdenken.

Jede Methode, deren Name eine Imperativform ist, hat jeweils mindestens ein Attribut des momentanen Zustands geändert, also beispielsweise move, turnLeft oder putCoin bei Klasse Robot in FopBot, setX und setY bei GeomShape2D.

Objektorientierte Abstraktion

TECHNISCHE UNIVERSITÄT DARMSTADT

- Objekte / Klassen sind die zentralen Bausteine.
  - Jede Subroutine gehört zu einer Klasse.
  - Heißt typischerweise Methode statt Subroutine / Prozedur / Funktion.
- Ein Objekt hat einen momentanen Zustand.
  - Ändert sich durch Schreibzugriffe auf seine Attribute und Aufrufe seiner Methoden.
  - Zeitliche Abläufe sind zu durchdenken.

Wir haben mit Fopbot erlebt, dass wir aufeinanderfolgende Zustandsänderungen genau durchdenken mussten. Vor allem wichtig und kompliziert wurde das bei Schleifen.

Objektorientierte Abstraktion

TECHNISCHE UNIVERSITÄT DARMSTADT

Objektorientiertes Programmdesign:

- **Die zu erstellende Funktionalität wird in Klassen und Interfaces zerlegt.**
- **Konzepte = Klassen und Interfaces.**
- **Programmablauf = Interaktion von Objekten.**
- **Durch Vererbung kann ein Konzept ein anderes erweitern, verfeinern oder variieren.**
  - ➢ **Objekte des abgeleiteten Konzeptes können an Stelle des Basiskonzepts verwendet werden.**

Das Paradigma, dem man beim Programmieren folgt, legt fest, wie man über das Gesamtprogramm denkt und wie man das Gesamtprogramm entwirft.

Objektorientierte Abstraktion

TECHNISCHE UNIVERSITÄT DARMSTADT

Objektorientiertes Programmdesign:

- **Die zu erstellende Funktionalität wird in Klassen und Interfaces zerlegt.**
- **Konzepte = Klassen und Interfaces.**
- **Programmablauf = Interaktion von Objekten.**
- **Durch Vererbung kann ein Konzept ein anderes erweitern, verfeinern oder variieren.**
  - ➢ **Objekte des abgeleiteten Konzeptes können an Stelle des Basiskonzepts verwendet werden.**

Im objektorientierten Programmentwurf denkt man von den Klassen her, denn das sind ja die zentralen Bausteine.

Bei Fopbot beginnt es mit Klasse Robot und Klasse World, und wir haben noch ein paar eigene Klassen von Robot abgeleitet. Auf Basis dieser Klassen haben wir dann Probleme gelöst wie etwa das Durchlaufen aller Felder eines Zimmers.

Objektorientierte Abstraktion

TECHNISCHE UNIVERSITÄT DARMSTADT

Objektorientiertes Programmdesign:

- **Die zu erstellende Funktionalität wird in Klassen und Interfaces zerlegt.**
- **Konzepte = Klassen und Interfaces.**
- **Programmablauf = Interaktion von Objekten.**
- **Durch Vererbung kann ein Konzept ein anderes erweitern, verfeinern oder variieren.**
  - ➢ **Objekte des abgeleiteten Konzeptes können an Stelle des Basiskonzepts verwendet werden.**

Jede Klasse und jedes Interface repräsentiert ein Konzept, das typischerweise auch den Namen der Klasse beziehungsweise des Interface bildet. Umgekehrt wird jedes abstrakte Konzept als eine Klasse oder ein Interface realisiert.

Bei Fopbot waren die Welt und der einzelne Roboter die grundlegenden Konzepte, auf denen alles aufbaute. Im geometrischen Fallbeispiel hatten wir das allgemeine Konzept einer zweidimensionalen geometrischen Figur durch die abstrakte Klasse GeomShape2D und konkrete geometrische Figuren durch Klassen wie Circle und Rectangle repräsentiert.

Objektorientierte Abstraktion

TECHNISCHE UNIVERSITÄT DARMSTADT

Objektorientiertes Programmdesign:

- **Die zu erstellende Funktionalität wird in Klassen und Interfaces zerlegt.**
- **Konzepte = Klassen und Interfaces.**
- **Programmablauf = Interaktion von Objekten.**
- **Durch Vererbung kann ein Konzept ein anderes erweitern, verfeinern oder variieren.**
  - ➢ **Objekte des abgeleiteten Konzeptes können an Stelle des Basiskonzepts verwendet werden.**

In Fopbot hatten vorrangig die Roboter mit der Welt interagiert und nur manchmal indirekt untereinander, nämlich wenn ein Roboter eine Münze ablegt und ein anderer diese Münze aufnimmt.

Objektorientierte Abstraktion

TECHNISCHE UNIVERSITÄT DARMSTADT

Objektorientiertes Programmdesign:

- Die zu erstellende Funktionalität wird in Klassen und Interfaces zerlegt.
- Konzepte = Klassen und Interfaces.
- Programmablauf = Interaktion von Objekten.
- Durch Vererbung kann ein Konzept ein anderes erweitern, verfeinern oder variieren.
  - ➢ Objekte des abgeleiteten Konzeptes können an Stelle des Basiskonzepts verwendet werden.

Offensichtlich repräsentiert eine abgeleitete Klasse ein Konzept, das mit dem Konzept der Basisklasse sehr viel gemein hat. Den hier farbig unterlegten Aspekt sehen wir uns auf der nächsten Folie genauer an.

Objektorientierte Abstraktion

TECHNISCHE UNIVERSITÄT DARMSTADT

Durch Vererbung ein Konzept erweitern, verfeinern oder variieren:

- **Robot erweitern: SymmTurner, FastRobot**
- **GeomShape2D verfeinern: Circle, Rectangle usw.**
- **SymmTurner variieren: SlowMotionRobot**

Hier sehen Sie repräsentative Beispiele aus der bisherigen Vorlesung aufgelistet.

Objektorientierte Abstraktion

TECHNISCHE UNIVERSITÄT DARMSTADT

Durch Vererbung ein Konzept erweitern, verfeinern oder variieren:

- **Robot erweitern: SymmTurner, FastRobot**
- **GeomShape2D verfeinern: Circle, Rectangle usw.**
- **SymmTurner variieren: SlowMotionRobot**

Diese zwei Klassen in Kapitel 01f haben jeweils zusätzliche Methoden gegenüber der Basisklasse Robot bekommen.

Objektorientierte Abstraktion

TECHNISCHE UNIVERSITÄT DARMSTADT

Durch Vererbung ein Konzept erweitern, verfeinern oder variieren:

- **Robot erweitern: SymmTurner, FastRobot**
- **GeomShape2D verfeinern: Circle, Rectangle usw.**
- **SymmTurner variieren: SlowMotionRobot**

Die Basisklasse GeomShape2D aus Kapitel 02 repräsentiert das allgemeine Konzept „zweidimensionale geometrische Form“. Jede Klasse, die wir davon abgeleitet hatten, repräsentierte eine spezielle Art davon.

Objektorientierte Abstraktion

TECHNISCHE UNIVERSITÄT DARMSTADT

Durch Vererbung ein Konzept erweitern, verfeinern oder variieren:

- Robot erweitern: SymmTurner, FastRobot
- GeomShape2D verfeinern: Circle, Rectangle usw.
- SymmTurner variieren: SlowMotionRobot

Bei Klasse SlowMotionRobot haben wir in Kapitel 01f etwas anderes gemacht als bei SymmTurner und FastRobot: Bei SlowMotionRobot haben wir nicht neue Methoden hinzugefügt, sondern ein paar schon in Robot beziehungsweise SymmTurner vorhandene Methoden – nämlich move, turnLeft und turnRight – überschrieben, so dass ihr Verhalten das ursprüngliche Verhalten dieser Methoden variiert.

TECHNISCHE UNIVERSITÄT DARMSTADT

Funktionale Abstraktion

Jetzt schauen wir uns also ein zweites Programmierparadigma an, das funktionale.

Die betrachteten Sprachen

TECHNISCHE UNIVERSITÄT DARMSTADT

- Diverse Konzepte von Java gehören zu den *funktionalen* Programmierkonzepten
- Diese Konzepte sind grundlegend für *funktionale* Programmiersprachen
- Wir betrachten und diskutieren funktionale Konzepte systematisch anhand Java und einer beispielhaft gewählten funktionalen Programmiersprache:

  HtDP-TL

  = Variante von Racket

  = Variante von Scheme

Klären wir erst einmal auf dieser Folie, was wir uns in diesem Kapitel eigentlich ansehen wollen.

Die betrachteten Sprachen

TECHNISCHE UNIVERSITÄT DARMSTADT

- Diverse Konzepte von Java gehören zu den *funktionalen* Programmierkonzepten
- Diese Konzepte sind grundlegend für *funktionale* Programmiersprachen
- Wir betrachten und diskutieren funktionale Konzepte systematisch anhand Java und einer beispielhaft gewählten funktionalen Programmiersprache:

  HtDP-TL

  = Variante von Racket

  = Variante von Scheme

Wir haben Java als eine objektorientierte Programmiersprache kennen gelernt. Aber das ist nicht die ganze Wahrheit. In heutigen Versionen von Java finden sich auch etliche Sprachkonstrukte, die man *funktional* nennt. In Java werden beide Welten miteinander in gewisser Weise vereint. Auf diese funktionalen Aspekte von Java werden wir hier auch zu sprechen kommen, auch wenn der Schwerpunkt auf einer anderen Sprache liegen wird.

Die betrachteten Sprachen

TECHNISCHE UNIVERSITÄT DARMSTADT

- Diverse Konzepte von Java gehören zu den *funktionalen* Programmierkonzepten
- Diese Konzepte sind grundlegend für *funktionale* Programmiersprachen
- Wir betrachten und diskutieren funktionale Konzepte systematisch anhand Java und einer beispielhaft gewählten funktionalen Programmiersprache:

  HtDP-TL

  = Variante von Racket

  = Variante von Scheme

Neben den objektorientierten gibt es auch andere Klassen von Programmiersprachen. Die funktionalen sind eine davon. Von hier sind die funktionalen Bestandteile von Java übernommen worden.

Die betrachteten Sprachen

TECHNISCHE UNIVERSITÄT DARMSTADT

- Diverse Konzepte von Java gehören zu den *funktionalen* Programmierkonzepten
- Diese Konzepte sind grundlegend für *funktionale* Programmiersprachen
- Wir betrachten und diskutieren funktionale Konzepte systematisch anhand Java und einer beispielhaft gewählten funktionalen Programmiersprache:

  HtDP-TL

  = Variante von Racket

  = Variante von Scheme

Was es damit nun auf sich hat, schauen wir uns anhand einer konkreten funktionalen Programmiersprache mit dem etwas sperrigen Namen HtDP-TL an. Scheme ist eine der bekannteren funktionalen Sprachen, davon ist Racket ein Dialekt, und davon wiederum ist HtDP-TL ein Dialekt.

Zur sprachlichen Vereinfachung reden wir von Racket, auch wenn das etwas ungenau ist.

Funktionales Programmieren

TECHNISCHE UNIVERSITÄT DARMSTADT

- Funktionen sind die zentralen Bausteine
  - $f : D_1 \times D_2 \times ... \times D_n \rightarrow R$
- Programmdesign:
  - Zerlegung der zu erstellenden Funktionalität in Funktionen
  - Funktionen rufen andere, grundlegendere Funktionen auf
- Funktionen werden variiert durch Parameter, die ihrerseits Funktionen sind

Was ist nun funktionales Programmieren? Eine grundsätzliche Antwort finden Sie auf dieser und den nächsten beiden Folien.

Funktionales Programmieren

TECHNISCHE UNIVERSITÄT DARMSTADT

- **Funktionen sind die zentralen Bausteine**
  - **f : D_1 x D_2 x ... x D_n → R**
- **Programmdesign:**
  - **Zerlegung der zu erstellenden Funktionalität in Funktionen**
  - **Funktionen rufen andere, grundlegendere Funktionen auf**
- **Funktionen werden variiert durch Parameter, die ihrerseits Funktionen sind**

Bei objektorientiertem Programmieren waren Referenztypen und ihre Objekte die zentralen Bausteine, und Methoden waren nur Bestandteile von Klassen. Wie Sie es aus der Mathematik kennen, sind Funktionen in funktionalen Sprachen unabhängige Entitäten, und zwar nicht irgendwelche Entitäten, sondern die Kernbausteine.

Funktionales Programmieren

TECHNISCHE UNIVERSITÄT DARMSTADT

- Funktionen sind die zentralen Bausteine
  - $f : D_1 \times D_2 \times \ldots \times D_n \rightarrow R$
- Programmdesign:
  - Zerlegung der zu erstellenden Funktionalität in Funktionen
  - Funktionen rufen andere, grundlegendere Funktionen auf
- Funktionen werden variiert durch Parameter, die ihrerseits Funktionen sind

So sind bei uns mathematische Funktionen gebildet: *n* Parameter und ein Rückgabewert. Die Zahl *n* kann beliebig sein und ist für jede Funktion ein fester Wert. Natürlich kann *n* auch gleich 0 sein, dann ist die Funktion konstant, da es ja keine Parameter gibt, von denen der Rückgabewert abhängt.

Die Definitionsbereiche sind hier mit D1 bis D*n* bezeichnet, D für englisch Domain, und R ist der Wertebereich der Funktion, R für Range.

Funktionales Programmieren

TECHNISCHE UNIVERSITÄT DARMSTADT

- Funktionen sind die zentralen Bausteine
  - $f : D_1 \times D_2 \times ... \times D_n \rightarrow R$
- Programmdesign:
  - Zerlegung der zu erstellenden Funktionalität in Funktionen
  - Funktionen rufen andere, grundlegendere Funktionen auf
- Funktionen werden variiert durch Parameter, die ihrerseits Funktionen sind

So wie bei objektorientiertem Programmieren die zu erstellende Funktionalität in Klassen zerlegt wird, wird sie im funktionalen Programmieren in Funktionen zerlegt.

Funktionales Programmieren

TECHNISCHE UNIVERSITÄT DARMSTADT

- Funktionen sind die zentralen Bausteine
  - $f : D_1 \times D_2 \times ... \times D_n \rightarrow R$
- Programmdesign:
  - Zerlegung der zu erstellenden Funktionalität in Funktionen
  - Funktionen rufen andere, grundlegendere Funktionen auf
- Funktionen werden variiert durch Parameter, die ihrerseits Funktionen sind

Und so wie bei objektorientiertem Programmieren die Ausführung eines Programms verstanden wird als Interaktion von Objekten, wird sie in funktionalem Programmieren verstanden als Interaktion von Funktionen in dem Sinne, dass eine Funktion die andere aufruft.

Funktionales Programmieren

TECHNISCHE UNIVERSITÄT DARMSTADT

- Funktionen sind die zentralen Bausteine
  - $f : D_1 \times D_2 \times ... \times D_n \rightarrow R$
- Programmdesign:
  - Zerlegung der zu erstellenden Funktionalität in Funktionen
  - Funktionen rufen andere, grundlegendere Funktionen auf
- Funktionen werden variiert durch Parameter, die ihrerseits Funktionen sind

Bei objektorientiertem Design hatten wir Variation von Klassen durch Vererbung. Hier haben wir Variation von Funktionen in anderer Form: Eine Funktion steuert den Rückgabewert einer anderen Funktion. Das geht dadurch, dass – wie wir sehen werden – Parameter einer Funktion wieder Funktionen sein können.

Deklaratives Programmieren

TECHNISCHE UNIVERSITÄT DARMSTADT

- Grundsätzlicher Gedanke:
  - Man schreibt nicht Befehle hin, die ausgeführt werden sollen
  - Nur die „Formel“ für das Ergebnis
- Das nennt man *deklarativen* Programmierstil
  - Im Gegensatz dazu *imperativer* Programmierstil in Java u.a. (z.B. C, C++)
- Konsequenzen:
  - Kein zeitlicher Ablauf
  - Keine Objektidentität

Um wirklich das Wesen des funktionalen Programmierens zu verstehen, müssen wir aber noch kurz eine Ebene höher gehen.

Deklaratives Programmieren

TECHNISCHE UNIVERSITÄT DARMSTADT

- Grundsätzlicher Gedanke:
  - Man schreibt nicht Befehle hin, die ausgeführt werden sollen
  - Nur die „Formel“ für das Ergebnis
- Das nennt man *deklarativen* Programmierstil
  - Im Gegensatz dazu *imperativer* Programmierstil in Java u.a. (z.B. C, C++)
- Konsequenzen:
  - Kein zeitlicher Ablauf
  - Keine Objektidentität

Die rein funktionalen Sprachen gehören zu einer noch größeren Sprachfamilie, den deklarativen Sprachen. Die Idee ist, dass man nicht schreibt, *wie* der Computer bei der Berechnung der Ergebnisse vorgehen soll, sondern nur, *was* er berechnen soll.

Deklaratives Programmieren

TECHNISCHE UNIVERSITÄT DARMSTADT

- Grundsätzlicher Gedanke:
  - Man schreibt nicht Befehle hin, die ausgeführt werden sollen
  - Nur die „Formel“ für das Ergebnis
- Das nennt man *deklarativen* Programmierstil
  - Im Gegensatz dazu *imperativer* Programmierstil in Java u.a. (z.B. C, C++)
- Konsequenzen:
  - Kein zeitlicher Ablauf
  - Keine Objektidentität

In diesen beiden Konsequenzen zeigen sich die großen Unterschiede zum objektorientierten Paradigma: Wir müssen keine zeitlichen Abläufe durchdenken, und wir müssen zum tieferen Verständnis auch keine Vorstellung davon entwickeln, wie das Ganze im Computerspeicher aussieht.

Das sind auch die wesentlichen Vorteile des funktionalen Paradigmas. Allerdings ergeben sich daraus wiederum andere intellektuelle Herausforderungen für Programmierer; Programmieren bleibt schwierig, nur die *Art* der Schwierigkeit ändert sich.

Deklaratives Programmieren

- Jeder Aufruf einer Funktion kann äquivalent ersetzt werden durch den Wert, den die Funktion für die Parameterwerte berechnet
- Insbesondere gilt: Egal wo die Funktion aufgerufen wird – dieselben Parameterwerte liefern dasselbe Ergebnis
- Funktionen haben nur Rückgabewerte, keine weiteren Effekte (= keine Seiteneffekte)
- Begriff: *referentielle Transparenz*

Kein zeitlicher Ablauf heißt, dass der Zeitpunkt des Aufrufs einer Funktion mit denselben Parameterwerten keine Rolle spielt, es kommt immer dasselbe Ergebnis heraus. Wenn man bei einem Aufruf die Parameterwerte weiß, könnte man den Aufruf dort einfach eliminieren und durch das Ergebnis der Funktion ersetzen, denn Funktionen haben nach der reinen funktionalen Lehre auch keine Seiteneffekte, also keine anderen Effekte als den Rückgabewert. Insbesondere wäre so etwas wie eine void-Methode, die nichts zurückliefert, im funktionalen Paradigma ziemlich unsinnig.

Der Fachbegriff dafür lautet *referentielle Transparenz*.

Kein zeitlicher Ablauf

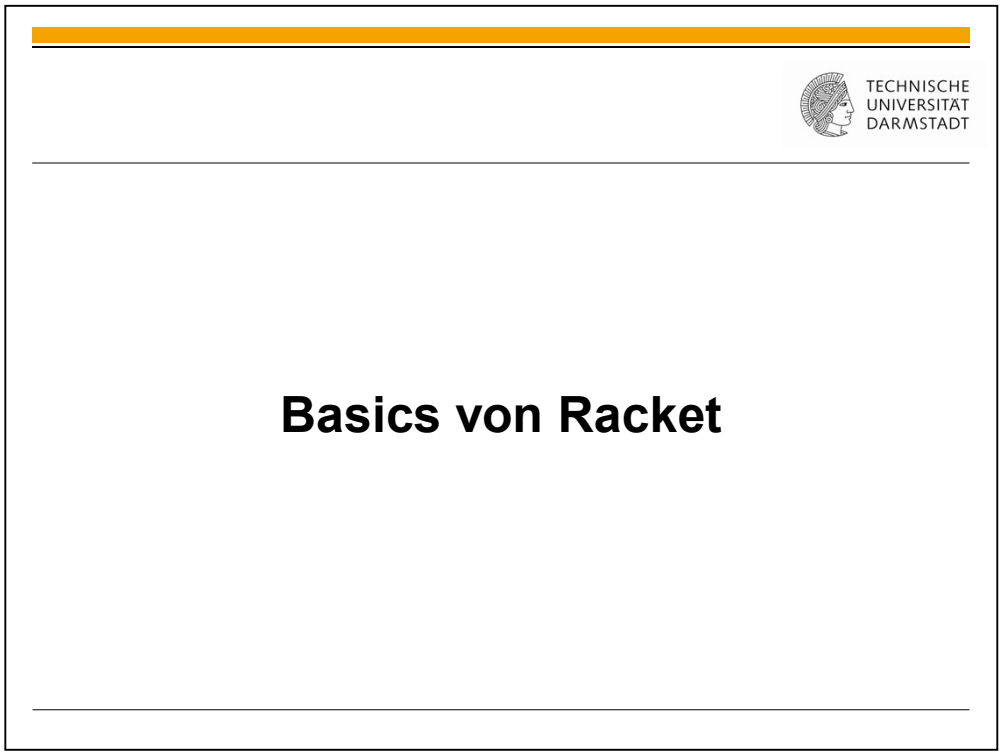
TECHNISCHE UNIVERSITÄT DARMSTADT

In Java hingegen:

```
int n = 10;                          X x = new Y();
System.out.println ( n );            x.m();
n = 11;                              x = new Z();
System.out.println ( n );            x.m();
```

- Derselbe Ausdruck kann zweimal völlig unterschiedliche Ergebnisse haben

In Java sieht es ja völlig anders aus: Wir haben in Java Variablen, also Bezeichner, die an einen Wert gebunden sind, aber dieser Wert kann sich im Ablauf des Programms ändern. In Racket haben wir nur Konstanten, und wann immer wir irgendwo eine Konstante einsetzen, können wir sicher sein, dass sie immer denselben Wert hat.



Wir kommen nun zur Sprache Racket.

Eine einfache(?) Methode

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public class Class1 {
    public static double add ( double x, double y ) {
        return x + y;
    }
}


double sum = Class1.add ( 2.71, 3.14 );
```

Wir gehen in diesem Kapitel bei jedem Einzelthema immer gleich vor: Wir schauen uns das Thema in Java an und sehen und staunen dann, wie einfach sich das jeweilige Thema in funktionalen Sprachen gestaltet.

Eine einfache(?) Methode

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public class Class1 {
    public static double add ( double x, double y ) {
        return x + y;
    }
}
```

```
double sum = Class1.add ( 2.71, 3.14 );
```

Unser erstes Beispiel ist eine mathematische Funktion, die zwei Zahlen bekommt und ihre Summe zurückliefert. Dafür müssen wir in Java eine ganze Menge Code drumherum schreiben, da eine solche Funktion nur als Methode einer Klasse realisierbar ist.

Eine einfache(?) Methode

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public class Class1 {
    public static double add ( double x, double y ) {
        return x + y;
    }
}
```

```
double sum = Class1.add ( 2.71, 3.14 );
```

Und wenn wir die Methode dann aufrufen, um die Summe zweier Zahlen in einer Variablen zu speichern, müssen wir den Klassennamen noch mit angeben, wenn wir nicht import static angewendet haben.

Nun mit beliebigen Zahlen

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public class Class2 {
    public static double add ( BigDecimal x, BigDecimal y ) {
        BigDecimal z = x.clone();
        z.add(y);
        return z;
    }
}

...

double sum = Class2.add ( new BigDecimal(2.71),
                          new BigDecimal(3.14) );
```

Die Methode add von Class1 hat aber noch eine Restriktion, die Zahlen in Racket nicht haben, wie wir gleich sehen werden: Der Typ double für die Parameter und den Rückgabewert kann zwar extrem große Zahlen mit extrem guter Genauigkeit darstellen, hat aber seine Grenzen, wie wir in Kapitel 01b, Abschnitt „Allgemein: Primitive Datentypen“ schon erfahren haben.

Nun mit beliebigen Zahlen

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public class Class2 {
    public static double add ( BigDecimal x, BigDecimal y ) {
        BigDecimal z = x.clone();
        z.add(y);
        return z;
    }
}

...

double sum = Class2.add ( new BigDecimal(2.71),
                          new BigDecimal(3.14) );
```

Um beliebig große und beliebig genaue Zahlen zu haben, können wir die Klasse BigDecimal aus dem Package java.math verwenden. Aber wie man sieht, wird damit alles noch einmal deutlich komplizierter als es mit double ohnehin schon war.

Vergleich mit Racket

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public class Class2 {
    public static double add ( BigDecimal x, BigDecimal y ) {
        BigDecimal z = x.clone();
        z.add(y);
        return z;
    }
}
```

```
( define ( add x y ) ( + x y ) )
```

Oben haben wir zum Vergleich noch einmal exakt wie auf der letzten Folie die Klasse Class2 mit BigDecimal statt double.

Vergleich mit Racket

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public class Class2 {
    public static double add ( BigDecimal x, BigDecimal y ) {
        BigDecimal z = x.clone();
        z.add(y);
        return z;
    }
}
```

```
( define ( add x y ) ( + x y ) )
```

Unten sehen wir jetzt, wie die Implementation einer solchen Funktion in Racket aussieht. Der Unterschied dürfte ins Auge springen.

Vergleich mit Racket

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public class Class2 {
    public static double add ( BigDecimal x, BigDecimal y ) {
        BigDecimal z = x.clone();
        z.add(y);
        return z;
    }
}
```

```
( define ( add x y ) ( + x y ) )
```

Mit dem Schlüsselwort define wird gesagt, dass jetzt irgendetwas definiert wird, entweder eine Konstante oder wie hier eine Funktion. Definitionen von Konstanten sehen wir in Kürze.

Vergleich mit Racket

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public class Class2 {
    public static double add ( BigDecimal x, BigDecimal y ) {
        BigDecimal z = x.clone();
        z.add(y);
        return z;
    }
}
```

```
( define ( add x y ) ( + x y ) )
```

Bei der Definition einer Funktion folgt direkt nach define ein Ausdruck in Klammern. Wir werden in Kürze sehen, dass die Klammern genau der Unterschied zur Definition einer Konstanten sind. Der Name der Funktion und die formalen Parameter werden in den Klammern einfach aufgezählt, ohne Komma oder anderes Trennsymbol, einfach nur durch Whitespace getrennt. Die Funktion heißt also add und hat zwei formale Parameter namens x und y.

Typnamen werden nicht hingeschrieben, dazu werden wir später noch mehr sagen. Die Identifier x und y sind also wirklich nur die *Namen* der formalen Parameter der Funktion add.

Vergleich mit Racket

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public class Class2 {
    public static double add ( BigDecimal x, BigDecimal y ) {
        BigDecimal z = x.clone();
        z.add(y);
        return z;
    }
}
```

```
( define ( add x y ) ( + x y ) )
```

Nach dem Klammerausdruck, der den Namen der Funktion und die formalen Parameter enthält, folgt als letzter Bestandteil die Implementation der Funktion. Ein return ist nicht notwendig, man schreibt nur den Ausdruck hin, dessen Wert zurückgeliefert werden soll.

Dies ist auch schon ein Beispiel für eine generelle Regel in Racket: Es kommt immer zuerst der Operator oder der Funktionsname, danach die Operanden beziehungsweise Parameter. Das nennt man Präfixnotation im Gegensatz zur üblichen Notation arithmetischer Ausdrücke, die Infixnotation heißt, weil der Operator *zwischen* die Operatoren geschrieben wird.

Vergleich mit Racket

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public class Class2 {
    public static double add ( BigDecimal x, BigDecimal y ) {
        BigDecimal z = x.clone();
        z.add(y);
        return z;
    }
}
```

```
( define ( add x y ) ( + x y ) )
```

Eine weitere generelle Regel sehen wir auch schon: Jede syntaktische Einheit, die nicht atomar ist, die also aus mehr als einem Identifier oder Literal zusammengesetzt ist, wird in Klammern gesetzt. Operatoren und Funktionen haben in Racket keine unterschiedlichen Bindungsstärken, sondern es müssen *immer alle* Klammern gesetzt werden.

Vergleich mit Racket

TECHNISCHE UNIVERSITÄT DARMSTADT

```
System.out.println ( Class1.add ( 2.71, 3.14 ) );

BigDecimal x = new BigDecimal ( 2.71 );
BigDecimal y = new BigDecimal ( 3.14 )
System.out.println ( Class2.add ( x, y ) );
```

```
( add 2.71 3.14 )
```

Auch der *Aufruf* einer Funktion gestaltet sich in funktionalen Sprachen sehr viel einfacher.

Vergleich mit Racket

TECHNISCHE UNIVERSITÄT DARMSTADT

```
System.out.println ( Class1.add ( 2.71, 3.14 ) );

BigDecimal x = new BigDecimal ( 2.71 );
BigDecimal y = new BigDecimal ( 3.14 )
System.out.println ( Class2.add ( x, y ) );
```

```
( add 2.71 3.14 )
```

So kompliziert ist es in Java schon mit dem primitiven Datentyp double.

Vergleich mit Racket

TECHNISCHE UNIVERSITÄT DARMSTADT

```
System.out.println ( Class1.add ( 2.71, 3.14 ) );
```

```
BigDecimal x = new BigDecimal ( 2.71 );
BigDecimal y = new BigDecimal ( 3.14 )
System.out.println ( Class2.add ( x, y ) );
```

```
( add 2.71 3.14 )
```

Und solche Verrenkungen muss man bei BigDecimal veranstalten.

Vergleich mit Racket

TECHNISCHE UNIVERSITÄT DARMSTADT

```
System.out.println ( Class1.add ( 2.71, 3.14 ) );

BigDecimal x = new BigDecimal ( 2.71 );
BigDecimal y = new BigDecimal ( 3.14 )
System.out.println ( Class2.add ( x, y ) );
```

```
( add 2.71 3.14 )
```

Im Gegensatz dazu ist der Aufruf in Racket sehr schlicht. Wieder dasselbe Muster wie bei der Definition der Funktion: Zuerst der Name der Funktion, dann die Parameter, alles zusammen in Klammern und ohne Trennsymbole, nur Whitespaces.

Vergleich mit Racket

TECHNISCHE UNIVERSITÄT DARMSTADT

```
System.out.println ( Class1.add ( 2.71, 3.14 ) );
```

```
BigDecimal x = new BigDecimal ( 2.71 );
BigDecimal y = new BigDecimal ( 3.14 )
System.out.println ( Class2.add ( x, y ) );
```

```
( add 2.71 3.14 )
```

Und wenn wir im System DrRacket einfach den Aufruf hinschreiben, wird das Ergebnis im Ausgabefenster auf den Bildschirm geschrieben, das heißt, der umständliche Aufruf zum Schreiben auf den Bildschirm entfällt auch noch.

Konstanten

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public class Math {
  ...
  public static final double PI = 3.14159:
  ...
}
```

```
( define my-pi 3.14159 )    ;  oder alternativ z.B.:
( define my-pi ( + 3 0.14159 ) )
```

Die Definition von globalen Konstanten ist sogar noch einfacher als die Definition von Funktionen.

Konstanten

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public class Math {
   ...
   public static final double PI = 3.14159:
   ...
}
```

```
( define my-pi 3.14159 )    ;  oder alternativ z.B.:
( define my-pi ( + 3 0.14159 ) )
```

In Java ist auch das etwas umständlich, da auch globale Konstanten zu Klassen gehören müssen.

Konstanten

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public class Math {
  ...
  public static final double PI = 3.14159:
  ...
}
```

```
( define my-pi 3.14159 )   ;  oder alternativ z.B.:
( define my-pi ( + 3 0.14159 ) )
```

In Racket hingegen ist die Definition von globalen Konstanten ganz einfach: erst Schlüsselwort define, dann der Name der Konstanten und schließlich der definierende Ausdruck, alles zusammen in Klammern.

Besteht der definierende Ausdruck nur aus einem einzigen Identifier oder Literal wie hier, dann wird er *nicht* in Klammern gesetzt.

Konstanten

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public class Math {
  ...
  public static final double PI = 3.14159:
  ...
}
```

```
( define my-pi 3.14159 )    ;  oder alternativ z.B.:
( define my-pi ( + 3 0.14159 ) )
```

Der definierende Ausdruck muss natürlich kein einfacher Ausdruck sein, sondern kann auch wieder ein zusammengesetzter Ausdruck sein. Wie schon gesagt, muss ein solcher zusammengesetzter Ausdruck im Racket *immer* in Klammern gesetzt sein.

Konstanten

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public class Math {
  ...
  public static final double PI = 3.14159:
  ...
}
```

```
( define my-pi 3.14159 )    ;  oder alternativ z.B.:
( define my-pi ( + 3 0.14159 ) )
```

Wie ebenfalls schon erwähnt, besteht der syntaktische Unterschied zur Definition einer Funktion, die keine Parameter hat, darin, dass der Name nicht in Klammern steht.

Wir nennen die Konstante nicht pi, sondern my-pi, weil die Konstante pi schon in Racket vordefiniert ist und dieser Identifier daher nicht noch einmal für eine Definition verwendet werden kann.

Konstanten

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public class Math {
  ...
  public static final double PI = 3.14159:
  ...
}
```

```
( define my-pi 3.14159 )   ; oder alternativ z.B.:
( define my-pi ( + 3 0.14159 ) )
```

Alles, was nach einem Semikolon noch in derselben Zeile kommt, ist auskommentiert, so wie beim doppelten Slash in Java.

Identifier (Bezeichner)

TECHNISCHE UNIVERSITÄT DARMSTADT

Was geht und was nicht geht:

- **Folgende Zeichen sind nicht erlaubt:**

  **( ) [ ] { } “ , ´ ` ; # | **

- **Whitespaces sind ebenfalls nicht erlaubt.**
- **Zahlen gehen nicht:**
  - **32 geht nicht.**
  - **a32 oder 3a2 oder 32a ginge.**
- **Ansonsten geht alles.**

Das sind die vollständigen Regeln in Racket, wie Identifier gebildet werden dürfen. Sie sehen, dass die Regeln sehr viel großzügiger als in Java sind.

***Vorgriff*: Insbesondere sind die arithmetischen Operatoren ebenfalls Identifier, aber schon vordefiniert und daher nicht für andere Zwecke nutzbar. Das heißt, arithmetische Operationen sind ganz normale Funktionen auf Zahlen. Wir werden später, in Kapitel 04c, sehen, dass man tatsächlich auch überall arithmetische Operationen einsetzen kann, wo Funktionen auf Zahlen erwartet werden.**

Identifier (Bezeichner)

TECHNISCHE UNIVERSITÄT DARMSTADT

Konventionen:

- **Keine Großbuchstaben**
- **Bindestriche zwischen einzelnen Wörtern**

this-identifier-conforms-to-all-conventions

this-2nd-identifier-does-so-as-well

Hier sehen Sie die wesentlichen Konventionen für Identifier. Offensichtlich sind diese ganz anders als die Konventionen in Java. Es ist typisch, dass verschiedene Programmiersprachen sich deutlich voneinander unterscheidende Konventionen haben.

Zahlen („number“) in Racket

TECHNISCHE UNIVERSITÄT DARMSTADT

- Exakte Zahlen:
  - ganzzahlig: 123
  - rational: 3/5
- Nichtexakte Zahlen: (sqrt 2)
- Komplexe Zahlen: 3.14159+3/5i

Verschiedene Arten von Zahlen werden in Racket zwar intern unterschiedlich repräsentiert, aber die Designer der Sprache haben versucht, die verschiedenen Zahlenarten für den Programmierer möglichst einheitlich darzustellen, so dass er sich mit den Unterschieden nur in bestimmten Situationen, in denen das unumgänglich ist, befassen muss.

Zahlen („number“) in Racket

TECHNISCHE UNIVERSITÄT DARMSTADT

- **Exakte Zahlen:**
  - ganzzahlig: 123
  - rational: 3/5
- Nichtexakte Zahlen: (sqrt 2)
- Komplexe Zahlen: 3.14159+3/5i

Soweit es irgend möglich ist, versucht Racket, Zahlen exakt darzustellen. Das Ergebnis einer arithmetischen Operation auf zwei exakt dargestellten Zahlen ist wieder eine exakt dargestellte Zahl.

Zahlen („number“) in Racket

TECHNISCHE UNIVERSITÄT DARMSTADT

- Exakte Zahlen:
  - ganzzahlig: 123
  - rational: 3/5
- Nichtexakte Zahlen: (sqrt 2)
- Komplexe Zahlen: 3.14159+3/5i

Im Gegensatz zu Java werden auch rationale Zahlen exakt dargestellt, nämlich durch zwei Zahlen, Zähler und Nenner. In diesem konkreten Fall werden also die beiden Zahlen 3 und 5 gespeichert und nicht der Zahlenwert 0.6, der im Binärsystem nicht exakt darstellbar ist, so wie etwa 1/7 im Dezimalsystem nicht exakt darstellbar ist.

Summe, Differenz, Produkt und Quotient zweier rationaler Zahlen werden in Racket wieder durch Zähler und Nenner exakt dargestellt.

Zahlen („number“) in Racket

TECHNISCHE UNIVERSITÄT DARMSTADT

- Exakte Zahlen:
  - ganzzahlig: 123
  - rational: 3/5
- Nichtexakte Zahlen: (sqrt 2)
- Komplexe Zahlen: 3.14159+3/5i

Irrationale Zahlen können nicht exakt dargestellt werden, sondern werden dann eben *nicht*exakt dargestellt.

Zahlen („number“) in Racket

TECHNISCHE UNIVERSITÄT DARMSTADT

- Exakte Zahlen:
  - ganzzahlig: 123
  - rational: 3/5
- Nichtexakte Zahlen: (sqrt 2)
- Komplexe Zahlen: 3.14159+3/5i

Racket macht bei reellen Zahlen nicht halt. Hinter einer Zahl kann sich auch eine komplexe Zahl verstecken. Eine komplexe Zahl besteht ihrerseits aus zwei Zahlen, dem Realteil und dem Imaginärteil, die jeweils exakt oder nichtexakt sein können. Das Kennzeichen eines komplexen Zahlenliterals ist das i unmittelbar nach dem Imaginärteil. Dank komplexer Zahlen kann man in Racket also beispielsweise auch Wurzeln aus negativen Zahlen ziehen.

Arithmetische Operationen

TECHNISCHE UNIVERSITÄT DARMSTADT

(+ 2 3)	5
(– 4 3)	1
(* 5 2)	10
(/ 37 30)	$1.2\overline{3}$
(modulo 20 3)	2

Sie sehen, dass die vier Grundrechenarten in Racket realisiert sind, aber in etwas anderer Schreibweise, eben in Präfixnotation: Der Operator steht nicht *zwischen* den beiden Operanden, sondern *davor*, und der ganze Ausdruck bestehend aus einem Operator und seinen Operanden muss zwingend in Klammern geschrieben werden. Rechts steht in jeder Zeile der Wert des jeweiligen Racket-Ausdrucks links.

Arithmetische Operationen

TECHNISCHE UNIVERSITÄT DARMSTADT

(+ 2 3)	5
(– 4 3)	1
(* 5 2)	10
(/ 37 30)	$1.2\overline{3}$
(modulo 20 3)	2

Hier sehen Sie noch einen speziellen Punkt aus der Schulmathematik: Division zweier Zahlen ergibt im Allgemeinen eine rationale Zahl, die mit einem Überstrich als periodische Dezimalzahl geschrieben wird.

Arithmetische Operationen

TECHNISCHE UNIVERSITÄT DARMSTADT

(+ 2 3)	5
(– 4 3)	1
(* 5 2)	10
(/ 37 30)	$1.2\overline{3}$
(modulo 20 3)	2

Der Rest bei der ganzzahligen Division hat kein eigenes Operatorsymbol, sondern ist eine eingebaute Funktion mit normalem Funktionsnamen.

***Vorgriff*: Es gibt noch eine Vielzahl von arithmetischen Funktionen wie modulo, von denen wir aber nur einige ausgewählte betrachten werden. Mit dem Verständnis, das Sie hier erwerben, sollten Sie keine Probleme damit haben, sich die weiteren arithmetischen Funktionen bei Bedarf mit Hilfe einer Racket-Doku selbst zu erschließen.**

Arithmetische Operationen

TECHNISCHE UNIVERSITÄT DARMSTADT

(+ 3.14 5)	8.14
(– 2.71 0)	2.71
(* 1.41 –2.71)	–3.8211
(/ –3.14 1.41)	–2.22695

In diesen weiteren Beispielen sehen Sie weitere nichtganze Zahlen, und Sie sehen auch, dass Zahlen negativ sein können, wie üblich angezeigt durch ein Minuszeichen vor dem Zahlenwert.

Arithmetische Operationen

TECHNISCHE UNIVERSITÄT DARMSTADT

(* (+ 2 3) 4)	20
(/ 4 (– 3 2))	4
(+ (* 3 4) (– 5 2))	15
(+ (+ (* 3 5) (* 4 6)) (/ 8 3))	$41.\overline{6}$
(+ 3 4 5)	12

Ausdrücke können wie in diesen Beispielen wiederum aus anderen Ausdrücken zusammengesetzt sein.

Arithmetische Operationen

TECHNISCHE UNIVERSITÄT DARMSTADT

(* (+ 2 3) 4)	20
(/ 4 (– 3 2))	4
(+ (* 3 4) (– 5 2))	15
(+ (+ (* 3 5) (* 4 6)) (/ 8 3))	$41.\overline{6}$
(+ 3 4 5)	12

Im ersten Beispiel ist der zweite Operand der Multiplikation wieder einfach nur eine Zahl, aber der erste Operand ist eine Addition zweier Zahlen. Zuerst werden 2 und 3 addiert, und das Ergebnis 5 wird dann mit 4 multipliziert.

Auch hier müssen alle Klammern zwingend hingeschrieben werden. Die generelle Regel zur Klammerung haben wir schon erwähnt: Ein atomarer Bestandteil wie ein Operator oder ein Identifier oder ein Literal wird *niemals* für sich allein in Klammern geschrieben, aber ein zusammengesetzter Ausdruck wie der erste Operand der Multiplikation hier wird *immer* in Klammern geschrieben.

Arithmetische Operationen

TECHNISCHE UNIVERSITÄT DARMSTADT

(* (+ 2 3) 4)	20
(/ 4 (– 3 2))	4
(+ (* 3 4) (– 5 2))	15
(+ (+ (* 3 5) (* 4 6)) (/ 8 3))	$41.\overline{6}$
(+ 3 4 5)	12

Im dritten Beispiel sind *beide* Operanden zusammengesetzt. Zuerst werden 3 mit 4 multipliziert und 2 von 5 abgezogen. Die beiden Zwischenergebnisse 12 und 3 werden danach addiert. Es gibt keine Punkt- vor Strichrechnung, auch die Multiplikation muss in Klammern gesetzt werden.

Arithmetische Operationen

TECHNISCHE UNIVERSITÄT DARMSTADT

(* (+ 2 3) 4)	20
(/ 4 (– 3 2))	4
(+ (* 3 4) (– 5 2))	15
(+ (+ (* 3 5) (* 4 6)) (/ 8 3))	$41.\overline{6}$
(+ 3 4 5)	12

Auch zusammengesetzte Ausdrücke können wieder aus zusammengesetzten Bestandteilen bestehen. Der erste Operand der äußeren Addition ist wiederum eine Addition, deren Operanden ebenfalls zusammengesetzt sind. Zuerst werden 3 mit 5 beziehungsweise 4 mit 6 multipliziert. Die Summe beider Teilergebnisse ergibt den ersten Operanden für die äußere Addition.

Natürlich könnte das immer so weitergehen, die Operanden der beiden Multiplikationen könnten ihrerseits wiederum beliebig komplex zusammengesetzt sein, und so weiter.

Arithmetische Operationen

TECHNISCHE UNIVERSITÄT DARMSTADT

(* (+ 2 3) 4)	20
(/ 4 (– 3 2))	4
(+ (* 3 4) (– 5 2))	15
(+ (+ (* 3 5) (* 4 6)) (/ 8 3))	$41.\overline{6}$
(+ 3 4 5)	12

Hier sehen wir erstmals, dass die Operatoren nicht unbedingt binär sind, also nicht unbedingt genau zwei Operanden haben müssen, sondern beliebig viele haben können.

Arithmetische Operationen

TECHNISCHE UNIVERSITÄT DARMSTADT

(+ 1 2 3 4)	1 + 2 + 3 + 4 = 10
(– 1 2 3 4)	1 – (2 + 3 + 4) = –8
(* 1 2 3 4)	1 * 2 * 3 * 4 = 24
(/ 1 2 3 4)	1 / (2 * 3 * 4) = 1/24

Auf dieser Folie sehen Sie überblicksweise die Ergebnisse, wenn einer der vier Operanden mehr als zwei Operatoren hat. Links sind vier beispielhafte Ausdrücke aufgeführt, je einer für jede der vier Grundrechenarten. Rechts ist jeweils in normaler schulmathematischer Notation das Ergebnis aufgeführt und wie es rechnerisch zustande kommt.

Arithmetische Operationen

TECHNISCHE UNIVERSITÄT DARMSTADT

(sqrt 4)	2
(sqrt 5)	#i2.236...
(sqrt (+ (* 3 4) (* 5 6)))	#i6.480...
pi	#i3.141...
(– e)	#i–2.718...

Diese Funktion haben wir schon einmal kurz gesehen. Die englische Abkürzung sqrt steht für square root, also Quadratwurzel, und genau das berechnet diese Funktion.

Arithmetische Operationen

TECHNISCHE UNIVERSITÄT DARMSTADT

(sqrt 4)	2
(sqrt 5)	#i2.236...
(sqrt (+ (* 3 4) (* 5 6)))	#i6.480...
pi	#i3.141...
(– e)	#i–2.718...

Wie Sie aus der Schulmathematik wissen, ist die Quadratwurzel der meisten nichtnegativen Zahlen irrational und daher grundsätzlich nicht als Zahlenwert exakt darstellbar. In Racket werden Rechenergebnisse, die nicht exakt darstellbar sind, mit einem Hashmark und einem i vor dem Zahlenwert dargestellt.

Arithmetische Operationen

TECHNISCHE UNIVERSITÄT DARMSTADT

(sqrt 4)	2
(sqrt 5)	#i2.236...
(sqrt (+ (* 3 4) (* 5 6)))	#i6.480...
pi	#i3.141...
(– e)	#i–2.718...

Wie schon gesagt: Die Kreiszahl pi ist in Racket vordefiniert. Da pi ebenfalls eine irrationale Zahl ist, steht auch hier Hashmark-i vor dem Zahlenwert von pi.

Arithmetische Operationen

TECHNISCHE UNIVERSITÄT DARMSTADT

(sqrt 4)	2
(sqrt 5)	#i2.236...
(sqrt (+ (* 3 4) (* 5 6)))	#i6.480...
pi	#i3.141...
(– e)	#i–2.718...

Die Eulersche Zahl ist ebenfalls vordefiniert. Wie Sie an diesem Beispiel sehen, steht das Minuszeichen einer nicht exakt darstellbaren negativen Zahl direkt hinter dem i, also direkt vor dem Zahlenwert.

Arithmetische Operationen

TECHNISCHE UNIVERSITÄT DARMSTADT

(sqrt 4)	2
(sqrt 5)	#i2.236...
(sqrt (+ (* 3 4) (* 5 6)))	#i6.480...
pi	#i3.141...
(– e)	#i–2.718...

Wenn Sie die Ausdrücke auf der linken Seite in DrRacket eingeben, dann bekommen Sie sehr viel mehr Nachkommastellen zu sehen als auf dieser Folie. Aus Gründen der Übersichtlichkeit werden hier nur drei Nachkommastellen gezeigt und die weiteren Stellen jeweils durch drei Punkte nur angedeutet.

Operationen mit ganzen Zahlen

TECHNISCHE UNIVERSITÄT DARMSTADT

```
( floor 3.14 )          ; 3
( ceiling 3.14 )        ; 4
( gcd 357 753 573 )     ; größter gemeinsamer Teiler
( modulo 753 357 )      ; Rest der ganzzahligen
                        ; Division 753 / 357
```

Noch ein paar Beispiele für vordefinierte mathematische Funktionen.

Operationen mit ganzen Zahlen

TECHNISCHE UNIVERSITÄT DARMSTADT

```
( floor 3.14 )          ; 3
( ceiling 3.14 )        ; 4
( gcd 357 753 573 )     ; größter gemeinsamer Teiler
( modulo 753 357 )      ; Rest der ganzzahligen
                        ; Division 753 / 357
```

Auf- und Abrunden auf die nächste ganze Zahl.

Operationen mit ganzen Zahlen

TECHNISCHE UNIVERSITÄT DARMSTADT

```
( floor 3.14 )          ; 3
( ceiling 3.14 )        ; 4
( gcd 357 753 573 )     ; größter gemeinsamer Teiler
( modulo 753 357 )      ; Rest der ganzzahligen
                        ; Division 753 / 357
```

Größter gemeinsamer Teiler, englisch greatest common denominator, von beliebig vielen ganzen Zahlen.

Operationen mit ganzen Zahlen

TECHNISCHE UNIVERSITÄT DARMSTADT

```
( floor 3.14 )          ; 3
( ceiling 3.14 )        ; 4
( gcd 357 753 573 )     ; größter gemeinsamer Teiler
( modulo 753 357 )      ; Rest der ganzzahligen
                        ; Division 753 / 357
```

Schon gesehen: das Analogon zum Prozentoperator in Java.

Boolesche Operationen

TECHNISCHE UNIVERSITÄT DARMSTADT

```
#t
#f

( and b1 b2 b3 )
( or b1 b2 b3 )
( not b )

( = x1 x2 x3 )    ;  ( and ( = x1 x2 ) ( = x2 x3 ) )
( < x1 x2 x3 )    ;  ( and ( < x1 x2 ) ( < x2 x3 ) )
( <= x1 x2 x3 )   ;  ( and ( <= x1 x2 ) ( <= x2 x3 ) )
```

Auch zwei boolesche Literale und diverse boolesche Operatoren sind in Racket vordefiniert.

Boolesche Operationen

TECHNISCHE UNIVERSITÄT DARMSTADT

```
#t
#f

( and b1 b2 b3 )
( or b1 b2 b3 )
( not b )

( = x1 x2 x3 )    ;  ( and ( = x1 x2 ) ( = x2 x3 ) )
( < x1 x2 x3 )    ;  ( and ( < x1 x2 ) ( < x2 x3 ) )
( <= x1 x2 x3 )   ;  ( and ( <= x1 x2 ) ( <= x2 x3 ) )
```

So sehen die Literale für true und false aus.

Boolesche Operationen

TECHNISCHE UNIVERSITÄT DARMSTADT

```
#t
#f

( and b1 b2 b3 )
( or b1 b2 b3 )
( not b )

( = x1 x2 x3 )    ; ( and ( = x1 x2 ) ( = x2 x3 ) )
( < x1 x2 x3 )    ; ( and ( < x1 x2 ) ( < x2 x3 ) )
( <= x1 x2 x3 )   ; ( and ( <= x1 x2 ) ( <= x2 x3 ) )
```

Die Verundung und die Veroderung können ebenfalls mehr als zwei Parameter haben, so wie wir das schon bei arithmetischen Operatoren gesehen hatten. Jeder dieser Parameter muss ein boolescher Ausdruck sein. Der Gesamtausdruck ist genau dann true, wenn bei and *jeder* der Parameter und bei or *mindestens einer* der Parameter true ist.

Boolesche Operationen

TECHNISCHE UNIVERSITÄT DARMSTADT

```
#t
#f

( and b1 b2 b3 )
( or b1 b2 b3 )
( not b )

( = x1 x2 x3 )    ;  ( and ( = x1 x2 ) ( = x2 x3 ) )
( < x1 x2 x3 )    ;  ( and ( < x1 x2 ) ( < x2 x3 ) )
( <= x1 x2 x3 )   ;  ( and ( <= x1 x2 ) ( <= x2 x3 ) )
```

Zur Verneinung braucht wohl nichts weiter gesagt zu werden.

Boolesche Operationen

TECHNISCHE UNIVERSITÄT DARMSTADT

```
#t
#f

( and b1 b2 b3 )
( or b1 b2 b3 )
( not b )

( = x1 x2 x3 )    ;  ( and ( = x1 x2 ) ( = x2 x3 ) )
( < x1 x2 x3 )    ;  ( and ( < x1 x2 ) ( < x2 x3 ) )
( <= x1 x2 x3 )   ;  ( and ( <= x1 x2 ) ( <= x2 x3 ) )
```

Auch die Vergleichsoperatoren können mehr als zwei Parameter haben. Sie ergeben in diesem Fall genau dann true, wenn der Vergleich zwischen je zwei aufeinanderfolgenden Parametern den Wert true ergibt.

Verzweigung

TECHNISCHE UNIVERSITÄT DARMSTADT

```
( define ( my-abs x ) ( if ( < 0 x ) x –x ) )

( define ( my-max x y ) ( if ( < x y ) y x ) )

( define ( sqr-of-max x y ) ( if ( < x y ) ( * y y ) ( * x x ) ) )

( define ( diff-is-integral x y )
      ( if ( integer? ( – x y ) ) #t #f ) )
```

Aus Java kennen Sie die if-Verzweigung und den Bedingungsoperator. Auch in Racket gibt es eine if-Verzweigung. Diese wird analog zum Bedingungsoperator in Java gebildet.

Verzweigung

TECHNISCHE UNIVERSITÄT DARMSTADT

```
( define ( my-abs x ) ( if ( < 0 x ) x –x ) )

( define ( my-max x y ) ( if ( < x y ) y x ) )

( define ( sqr-of-max x y ) ( if ( < x y ) ( * y y ) ( * x x ) ) )

( define ( diff-is-integral x y )
      ( if ( integer? ( – x y ) ) #t #f ) )
```

Auch wenn es nicht ganz der Systematik hinter Racket entspricht: Sie können sich die if-Verzweigung in Racket als eine boolesche Funktion mit drei Parametern vorstellen. Der Name dieser Funktion ist einfach: „if“.

Verzweigung

TECHNISCHE UNIVERSITÄT DARMSTADT

```
( define ( my-abs x ) ( if ( < 0 x ) x –x ) )

( define ( my-max x y ) ( if ( < x y ) y x ) )

( define ( sqr-of-max x y ) ( if ( < x y ) ( * y y ) ( * x x ) ) )

( define ( diff-is-integral x y )
      ( if ( integer? ( – x y ) ) #t #f ) )
```

Der erste Parameter muss boolesch sein und fungiert als die Bedingung der if-Verzweigung.

Verzweigung

TECHNISCHE UNIVERSITÄT DARMSTADT

```
( define ( my-abs x ) ( if ( < 0 x ) x –x ) )

( define ( my-max x y ) ( if ( < x y ) y x ) )

( define ( sqr-of-max x y ) ( if ( < x y ) ( * y y ) ( * x x ) ) )

( define ( diff-is-integral x y )
      ( if ( integer? ( – x y ) ) #t #f ) )
```

Die vordefinierte boolesche Funktion integer? haben wir bisher noch nicht gesehen. Sie hat einen Parameter und liefert genau dann true zurück, wenn der Parameter eine ganze Zahl ist.

Beachten Sie, dass es hier – anders als in Java – nicht um formale Typen, sondern wirklich um deren Werte geht. So könnten x und y beliebige exakt darstellbare, also rationale oder sogar komplexe Zahlen mit rationalem Real- und Imaginärteil sein: Sofern ihre Differenz eine ganze Zahl ist, wird true zurückgeliefert.

***Vorgriff*: Auf der nächsten Folie sehen wir uns diese und analoge Funktionen systematisch an.**

Verzweigung

TECHNISCHE UNIVERSITÄT DARMSTADT

```
( define ( my-abs x ) ( if ( < 0 x ) x –x ) )

( define ( my-max x y ) ( if ( < x y ) y x ) )

( define ( sqr-of-max x y ) ( if ( < x y ) ( * y y ) ( * x x ) ) )

( define ( diff-is-integral x y )
      ( if ( integer? ( – x y ) ) #t #f ) )
```

Der zweite Parameter der Funktion if ist der Rückgabewert der if-Funktion in dem Fall, dass der erste Parameter true zurückliefert. Das ist also völlig analog zum Bedingungsoperator in Java.

Verzweigung

TECHNISCHE UNIVERSITÄT DARMSTADT

```
( define ( my-abs x ) ( if ( < 0 x ) x –x ) )

( define ( my-max x y ) ( if ( < x y ) y x ) )

( define ( sqr-of-max x y ) ( if ( < x y ) ( * y y ) ( * x x ) ) )

( define ( diff-is-integral x y )
      ( if ( integer? ( – x y ) ) #t #f ) )
```

Und ebenso analog zum Bedingungsoperator in Java ist, dass es – im Gegensatz zur if-Verzweigung in Java – unbedingt eine Art else-Teil geben muss. Das ist der dritte Parameter: Sein Wert ist der Rückgabewert der gesamten if-Funktion in dem Fall, dass der erste Parameter false ergibt.

Verzweigung

TECHNISCHE UNIVERSITÄT DARMSTADT

```
( define ( my-abs x ) ( if ( < 0 x ) x –x ) )

( define ( my-max x y ) ( if ( < x y ) y x ) )

( define ( sqr-of-max x y ) ( if ( < x y ) ( * y y ) ( * x x ) ) )

( define ( diff-is-integral x y )
     ( if ( integer? ( – x y ) ) #t #f ) )
```

Wie bei jeder anderen Funktion in Racket auch, muss auch die if-Verzweigung in Klammern aufgerufen werden, da ja der Name der Funktion und die Parameter einen zusammengesetzten, nicht atomaren Ausdruck bilden.

Boolesche Operationen

TECHNISCHE UNIVERSITÄT DARMSTADT

```
( number? x )
( real? x )
( rational? x )
( integer? x )
( natural? x )
( symbol? x )
( string? x )
```

Wie soeben versprochen, kommen wir nun zur booleschen Funktion integer? und ihren Verwandten.

Boolesche Operationen

TECHNISCHE UNIVERSITÄT DARMSTADT

```
( number? x )
( real? x )
( rational? x )
( integer? x )
( natural? x )
( symbol? x )
( string? x )
```

Wir haben schon gesehen, dass Identifier in Racket sehr viel flexibler gebildet werden dürfen als in Java. Unter anderem dürfen sie auch Fragezeichen enthalten. Gerade für boolesche Abfragen ist das natürlich sinnvoll.

Der hier farblich unterlegte Aufruf liefert genau dann true, wenn x ein Zahlenwert ist, egal von welcher Art. Im Zahlenmodell von Racket umfasst die Menge der komplexen Zahlen alle anderen Zahlenmengen. Daher kann man auch sagen, dass diese Funktion abprüft, ob ihr Parameter eine komplexe Zahl ist.

Boolesche Operationen

TECHNISCHE UNIVERSITÄT DARMSTADT

```
( number? x )
( real? x )
( rational? x )
( integer? x )
( natural? x )
( symbol? x )
( string? x )
```

Diese Abfrage liefert true, wenn x eine Zahl, aber keine echt komplexe Zahl ist, also wenn die Zahl keinen Imaginärteil hat. Mathematisch kann man auch sagen: wenn ihr Imaginärteil exakt 0 ist.

Boolesche Operationen

TECHNISCHE UNIVERSITÄT DARMSTADT

(number? x)

(real? x)

(rational? x)

(integer? x)

(natural? x)

(symbol? x)

(string? x)

Wenn x eine rationale Zahl ist, wozu natürlich auch die ganzen Zahlen gehören, dann liefert diese Abfrage true.

Boolesche Operationen

TECHNISCHE UNIVERSITÄT DARMSTADT

```
( number? x )
( real? x )
( rational? x )
( integer? x )
( natural? x )
( symbol? x )
( string? x )
```

Den Test auf Ganzzahligkeit hatten wir schon im vorhergehenden Beispiel.

Boolesche Operationen

TECHNISCHE UNIVERSITÄT DARMSTADT

```
( number? x )
( real? x )
( rational? x )
( integer? x )
( natural? x )
( symbol? x )
( string? x )
```

Manchmal ist es nicht nur wichtig zu wissen, ob ein Wert ganzzahlig ist, sondern ob er darüber hinaus nichtnegativ, also eine natürliche Zahl ist. Das kann man natürlich auch durch Verundung zweier boolescher Abfragen realisieren: Test auf Ganzzahligkeit und Test auf Nichtnegativität. Aber dieser Fall ist so häufig, dass eine eigene Abfrage auf Natürlichzahligkeit sinnvoll ist. Sie ist auch deshalb sinnvoll, weil sie leichter für Leser des Quelltextes zu erfassen ist: Die Abfrage natural? sagt klipp und klar, worum es geht, während man dafür die oben erwähnte Verundung als Leser erst durchdenken müsste.

Boolesche Operationen

TECHNISCHE UNIVERSITÄT DARMSTADT

```
( number? x )
( real? x )
( rational? x )
( integer? x )
( natural? x )
( symbol? x )
( string? x )
```

Der boolesche Test, ob der Parameter ein Symbol ist, wird analog gebildet.

Boolesche Operationen

TECHNISCHE UNIVERSITÄT DARMSTADT

```
( number? x )
( real? x )
( rational? x )
( integer? x )
( natural? x )
( symbol? x )
( string? x )
```

Strings sind ungefähr dasselbe wie in Java, nur dass sie in Racket eingebaut sind, während es in Java eine Klasse String gibt. String-Literale sind wie in Java in Hochkommas gesetzt. In den genauen Zugriffsmöglichkeiten und in der Syntax gibt es natürlich kleine Unterschiede. Hier wird also abgefragt, ob x ein String ist. Mehr werden wir zu Strings in Racket nicht sagen, Strings sind nicht unser Fokus.

Symbole

TECHNISCHE UNIVERSITÄT DARMSTADT

```
( define last-name ´Spielberg )

( if ( symbol=? last-name ´Lucas ) ( ..... ) ( ..... ) )
```

Das ist eine Art von Datentyp, die es in Java nicht gibt. Im Prinzip sind Symbole Identifier mit einem Hochkomma vorneweg.

Der Bedeutungsunterschied zwischen Identifiern und Symbolen ist aber fundamental: Ein Identifier steht in Racket für eine Konstante oder eine Funktion. In beiden Fällen wird der Name durch ein define an den Wert der Konstanten beziehungsweise an die Implementation der Funktion gebunden.

Ein Symbol hingegen steht für nichts, nur für sich selbst. Es hat sicherlich für den Programmierer eine Bedeutung, aber nicht für Racket.

Symbole

TECHNISCHE UNIVERSITÄT DARMSTADT

```
( define last-name ´Spielberg )

( if ( symbol=? last-name ´Lucas ) ( ..... ) ( ..... ) )
```

Das Gleichheitszeichen und das Fragezeichen gehören zum Namen dieser vordefinierten booleschen Funktion. Sie ist auf Symbole anwendbar und liefert genau dann true, wenn erstens beide Parameter Symbole sind und zweitens die beiden Symbole zudem gleich sind.

Viel mehr als Test auf Gleichheit oder Ungleichheit kann man mit Symbolen auch nicht machen, da Symbole für Racket eben keine Bedeutung haben.

Typ einer Funktion

TECHNISCHE UNIVERSITÄT DARMSTADT

```
;; Type: number number -> number
;;
;; Returns: the sum of the two parameters

( define  ( add x y )  ( + x y )  )
```

Wir haben mehrfach gesehen, dass wir gar keine Typen angeben für die Werte, mit denen wir so arbeiten. Tatsächlich wird in Racket erst zur Laufzeit geprüft, ob die Typen der Operanden einer Operation zu dieser Operation passen. Und wenn das nicht der Fall ist, bricht DrRacket die Ausführung des Programms mit einer entsprechenden Fehlermeldung ab.

Typ einer Funktion

TECHNISCHE UNIVERSITÄT DARMSTADT

```
;; Type: number number -> number
;;
;; Returns: the sum of the two parameters

( define  ( add x y )  ( + x y ) )
```

Für den Nutzer einer Funktion sollten wir daher unbedingt etwas tun, was in Java nicht nötig ist: in einem Kommentar vertraglich zusichern, welche Typen die Parameter haben und welcher Typ zurückgeliefert wird.

Dieses Beispiel folgt der allgemeinen Konvention für Racket: eine Aufzählung der Parametertypen in der selben Reihenfolge, wie sie bei der Definition und beim Aufruf der Funktion auftreten, nach einem Pfeil dann der Rückgabetyp, das alles wieder ohne Kommas oder ähnliches.

Halten wir fest, dass jede Funktion einen Typ hat, der aus den Typen der Parameter und dem Rückgabetyp gebildet wird.

Typ einer Funktion

TECHNISCHE UNIVERSITÄT DARMSTADT

```
;; Type: number number -> number
;;
;; Returns: the sum of the two parameters

( define  ( add x y )  ( + x y )  )
```

Außerdem sollten wir zu jeder Funktion – und sei sie noch so einfach – in einem Kommentar dazuschreiben, was der Wert eigentlich beinhaltet, der zurückgeliefert wird. das ist bei Java natürlich genauso.

Typ einer Funktion

TECHNISCHE UNIVERSITÄT DARMSTADT

```
;; Type: number number -> number
;;
;; Returns: the sum of the two parameters

( define  ( add x y )  ( + x y )  )
```

→ Vertrag von add

Was wir hier oben in einen Kommentar schreiben, werden wir von jetzt an den *Vertrag* der Funktion nennen. Der Begriff rührt von der Vorstellung her: Wenn der Nutzer der Funktion seinen Teil des Vertrags erfüllt, nämlich zwei Zahlen als aktuale Parameter einzugeben, dann erfüllt die Funktion ihren Teil des Vertrags, nämlich eine Zahl zurückzuliefern, deren Wert gleich der Summe der beiden aktualen Parameterwerte ist.

***Nebenbemerkung*: Im Zusammenhang mit Racket versteht man unter „Vertrag“ eher das, was oben mit „Type“ eingeleitet ist. Wie wir hier das Wort „Vertrag“ benutzen, passt aber gut dazu, wie „Vertrag“ allgemein in der Informatik verstanden wird.**

Typ einer Funktion

TECHNISCHE UNIVERSITÄT DARMSTADT

```
;; Type: number number -> number
;;
;; Returns: the sum of the two parameters

( define  ( add x y )  ( + x y )  )
```

Es ist allgemein anerkannte Konvention, dass ein Kommentar, der die ganze Zeile umfasst, mit zwei Semikolons beginnt, auch wenn ein einzelnes Semikolon für die Auskommentierung ausreicht und das zweite Semikolon gar keine Funktion hat, sondern für DrRacket schon Teil des Kommentars ist.

Laufzeitchecks

TECHNISCHE UNIVERSITÄT DARMSTADT

```
;; Type: number number -> number
;; Precondition: the second parameter must not
;;                    equal zero
;; Returns: x divided by y
(  define  ( divide x y )  ( / x y )  )

(  check-expect  ( divide 15 3 )  5  )
(  check-within  ( divide pi e )  1.15  0.01  )
(  check-error  ( divide 15 0 )  "/: division by zero"  )
```

Jetzt ein ergänzendes Thema zu Funktionen in Racket: Racket bietet Möglichkeiten, die Korrektheit von Funktionen zur Laufzeit zu testen.

Laufzeitchecks

TECHNISCHE UNIVERSITÄT DARMSTADT

```
;; Type: number number -> number
;; Precondition: the second parameter must not
;;                    equal zero
;; Returns: x divided by y
(  define  ( divide x y )  ( / x y )  )

(  check-expect  ( divide 15 3 )  5  )
(  check-within  ( divide pi e )  1.15  0.01  )
(  check-error  ( divide 15 0 )  "/: division by zero"  )
```

Die Funktion, die wir hier beispielhaft testen, dient nur der Illustration. Sie ist insofern gut gewählt, als sie auch einen möglichen Fehlerfall beinhaltet, nämlich Division durch 0.

Laufzeitchecks

TECHNISCHE UNIVERSITÄT DARMSTADT

```
;; Type: number number -> number
;; Precondition: the second parameter must not
;;                    equal zero
;; Returns: x divided by y
(  define  ( divide x y )  ( / x y )  )

(  check-expect  ( divide 15 3 )  5  )
(  check-within  ( divide pi e )  1.15  0.01  )
(  check-error  ( divide 15 0 )  "/: division by zero"  )
```

Die vordefinierte Funktion check-expect bekommt zwei Ausdrücke als Parameter und liefert eine Fehlermeldung, falls die beiden Ausdrücke nicht denselben Wert haben. Die farblich unterlegte Zeile testet also, ob Funktion divide mit den aktualen Parametern 15 und 3 tatsächlich wie erwartet 5 als Rückgabewert hat.

Wenn der Test gelingt, läuft die Ausführung des Programms weiter, ansonsten wird sie mit einer entsprechenden Fehlermeldung abgebrochen.

Selbstverständlich kann man beliebig viele Tests aller drei Arten beliebig gemischt definieren.

Laufzeitchecks

TECHNISCHE UNIVERSITÄT DARMSTADT

```
;; Type: number number -> number
;; Precondition: the second parameter must not
;;                    equal zero
;; Returns: x divided by y
(  define  ( divide x y )  ( / x y )  )

(  check-expect  ( divide 15 3 )  5  )
(  check-within  ( divide pi e )  1.15  0.01  )
(  check-error  ( divide 15 0 )  "/: division by zero"  )
```

Wollten wir das Ergebnis von divide mit zwei nicht exakt darstellbaren Zahlen wie pi und e genauso mit check-expect testen, dann müsste der zweite Parameter von check-expect auf sämtliche dargestellten Nachkommastellen genau gleich dem Ergebnis der Division sein, sonst sind die beiden aktualen Parameter ungleich und der Test schlägt fehl. Das ist natürlich umständlich und nur in so einfachen Fällen wie hier überhaupt möglich. In der Praxis wird man selten das exakte Ergebnis einer Funktion auf anderem Wege nochmals bestimmen können, um die beiden Werte miteinander zu vergleichen.

Was man aber in der Regel haben kann, ist ein Schätzwert für das Ergebnis. Mit check-within kann man testen, ob zwei Zahlen ausreichend nahe beieinander liegen, um als gleich angesehen zu werden. Die ersten beiden Parameter sind die beiden zu vergleichenden Werte. Der Test liefert eine Fehlermeldung und Prozessabbruch, falls die Werte der ersten beiden Parameter sich um mehr als den dritten Parameter unterscheiden.

Laufzeitchecks

TECHNISCHE UNIVERSITÄT DARMSTADT

```
;; Type: number number -> number
;; Precondition: the second parameter must not
;;                    equal zero
;; Returns: x divided by y
(  define  ( divide x y )  ( / x y )  )

(  check-expect  ( divide 15 3 )  5  )
(  check-within  ( divide pi e )  1.15  0.01  )
(  check-error  ( divide 15 0 )  "/: division by zero"  )
```

Der typische Anwendungsfall ist hier gezeigt: Der Rückgabewert einer Funktion wird mit dem Schätzwert verglichen, den wir dafür auf anderem Wege berechnet haben. Natürlich sollte der dritte Parameter so klein wie möglich gewählt werden, um fälschlich positive Tests möglichst zu vermeiden. Aber *wie* klein möglich ist, hängt vom konkreten Fall und seinen Umständen ab.

Laufzeitchecks

TECHNISCHE UNIVERSITÄT DARMSTADT

```
;; Type: number number -> number
;; Precondition: the second parameter must not
;;                    equal zero
;; Returns: x divided by y
(  define  ( divide x y )  ( / x y )  )

(  check-expect  ( divide 15 3 )  5  )
(  check-within  ( divide pi e )  1.15  0.01  )
(  check-error  ( divide 15 0 )  "/: division by zero"  )
```

Es gehört durchaus zur Korrektheit der Funktion divide, dass tatsächlich eine Fehlermeldung ausgegeben wird, falls der Divisor gleich 0 ist. Ein Test mit check-error gelingt, wenn der erste aktuale Parameter einen Fehler liefert und wenn dieser Fehler gleich dem zweiten aktualen Parameter ist.

Wir sehen, dass Fehlermeldungen Strings sind, die wie in Java in doppelte Hochkommas gesetzt sind. In der Dokumentation zu Racket schlägt man nach, wie die Fehlermeldung genau lautet, und setzt diese als zweiten aktualen Parameter ein.

Laufzeitchecks

TECHNISCHE UNIVERSITÄT DARMSTADT

```
;; Type: number number -> number
;; Precondition: the second parameter must not
;;                    equal zero
;; Returns: x divided by y
```

→ Vertrag von divide

Das ist eine übliche Form eines Vertrags, denn sehr häufig reicht es nicht, vom Nutzer der Methode zu fordern, dass die aktualen Parameter von den erwarteten Typen sind, wie sie in der Type-Klausel spezifiziert sind. Häufig müssen einzelne oder alle aktualen Parameterwerte noch weitere Vorbedingungen erfüllen.

Laufzeitchecks

TECHNISCHE UNIVERSITÄT DARMSTADT

```
;; Type: number number -> number
;; Precondition: the second parameter must not
;;                    equal zero
;; Returns: x divided by y

(  define  ( divide x y )
    ( if ( = y 0 ) ( error “Division by 0“ ) ( / x y )  )  )
```

Man kann auch Tests in eine Funktion einbauen. Ein solcher Test wird dann im Gegensatz zu check-expect, check-within und check-error nicht einfach nur einmal mit vorgegebenen Parametern ausgeführt, sondern bei jedem Aufruf von divide mit den jeweiligen aktualen Parameterwerten.

Dazu gibt es verschiedene Möglichkeiten in Racket. Hier sehen Sie die einfachste: eine vordefinierte Funktion namens error mit einem Parameter, der ein String sein muss. Diesen String können wir frei wählen (im Gegensatz zur Situation bei check-error auf der letzten Folie).

Sollte die Funktion error aufgerufen werden – sollte im Beispiel also y gleich 0 sein –, dann bricht das Programm mit einer Fehlermeldung ab, in deren genauen Wortlaut der String eingearbeitet ist.

Komplexere Berechnung

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public class Class3 {
    public static double euclid2 ( double x, double y ) {
        return Math.sqrt ( x * x + y * y );
}

...

;; Type: real real -> real
(  define  ( euclid2 x y )  ( sqrt ( + ( * x x ) ( * y y ) ) ) )
```

Von jetzt an schreiben wir zu jeder Funktion immer ihren Typ hinzu.

Komplexere Berechnung

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public class Class3 {
    public static double euclid2 ( double x, double y ) {
        return Math.sqrt ( x * x + y * y );
}

...

;; Type: real real -> real
(  define  ( euclid2 x y )  ( sqrt ( + ( * x x ) ( * y y ) ) ) )
```

Wir betrachten als nächstes eine etwas komplexere Operation: die Euklidische Norm eines zweidimensionalen Punktes.

Komplexere Berechnung

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public class Class3 {
    public static double euclid2 ( double x, double y ) {
        return Math.sqrt ( x * x + y * y );
}

...

;; Type: real real -> real
(  define  ( euclid2 x y )  ( sqrt ( + ( * x x ) ( * y y ) ) ) )
```

Laut Typ erwartet die Funktion euclid2 vom Nutzer, dass die Parameter x und y reelle Zahlen sind, und garantiert dem Nutzer in diesem Fall, dass das Ergebnis ebenfalls eine reelle Zahl ist.

***Nebenbemerkung*: Die Funktion euclid2 könnte problemlos auch komplexe Zahlen verarbeiten. Aber für Vektoren auf komplexen Zahlen ist die Euklidische Norm leicht anders definiert (fällt bei reellen Zahlen aber mit der Definition der Euklidischen Norm für Vektoren auf reellen Zahlen zusammen). Es wäre daher verwirrend, die wie auf dieser Folie definierte Funktion euclid2 für komplexe Zahlen anzubieten, daher stehen im Typ nur reelle Zahlen.**

Komplexere Berechnung

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public class Class3 {
    public static double euclid2 ( double x, double y ) {
        return Math.sqrt ( x * x + y * y );
}

...

;; Type: real real -> real
(  define  ( euclid2 x y )  ( sqrt ( + ( * x x ) ( * y y )  )  )  )
```

Die Operanden für die beiden Multiplikationen sind die Parameter der Funktion euclid2, also einfache Ausdrücke. Die beiden Multiplikationen sind wiederum Operanden der Addition. Diese beiden Operanden sind zusammengesetzt, müssen also in Klammern sein.

Komplexere Berechnung

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public class Class3 {
    public static double euclid2 ( double x, double y ) {
        return Math.sqrt ( x * x + y * y );
}

...

;; Type: real real -> real
(  define  ( euclid2 x y )  ( sqrt ( + ( * x x ) ( * y y ) ) ) )
```

Die ganze Addition bildet wiederum den Parameter für die vordefinierte Funktion sqrt, die eine einzelne Zahl als Parameter erwartet. Die Addition ist auch wieder ein zusammengesetzter Ausdruck, also in Klammern.

Komplexere Berechnung

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public class Class3 {
    public static double euclid2 ( double x, double y ) {
        return Math.sqrt ( x * x + y * y );
}

...

;; Type: real real -> real
(  define  ( euclid2 x y )  ( sqrt ( + ( * x x ) ( * y y ) ) ) )
```

Und der Aufruf der Quadratwurzelfunktion mit einem Parameter ist ebenfalls ein zusammengesetzter Ausdruck, also in Klammern.

TECHNISCHE UNIVERSITÄT DARMSTADT

(Idealisiertes) Objektmodell

Nach diesen ersten Basics klären wir einen grundlegenden Unterschied zwischen Racket und Java, der durchaus typisch ist für den Unterschied zwischen deklarativen und imperativen Sprachen.

Wir werden gleich etwas zu dem in Klammern gesetzten Attribut „Idealisiertes“ sagen.

(Idealisiertes) Objektmodell

TECHNISCHE UNIVERSITÄT DARMSTADT

- Es gibt keine Objekte, nur Werte
  - Werte sind *immer*(?) Konstante, *nie*(?) Variable
  - Werte werden *immer* kopiert
- Laufzeitsystem kann intern zur Optimierung von dieser Grundlogik abweichen
  - Ist für Programmierer nicht sichtbar

```
( define my-const-1 12345 )
( define  my-const-2 my-const-1 )
( define ( my-fct my-const ) ( ..... ) )
```

Oben sehen Sie die Grundprinzipien des Objektmodells, die Beispiele unten nehmen wir zur Illustration.

(Idealisiertes) Objektmodell

TECHNISCHE UNIVERSITÄT DARMSTADT

- Es gibt keine Objekte, nur Werte
  - Werte sind *immer*(?) Konstante, *nie*(?) Variable
  - Werte werden *immer* kopiert
- Laufzeitsystem kann intern zur Optimierung von dieser Grundlogik abweichen
  - Ist für Programmierer nicht sichtbar

```
( define my-const-1 12345 )
( define  my-const-2 my-const-1 )
( define ( my-fct my-const ) ( ..... ) )
```

Wichtig ist zu verstehen, dass Werte niemals modifiziert, also durch neue Werte überschrieben werden; es gibt nur Konstanten, keine Variablen.

Wenn ein neuer Wert in Form einer Konstante eingerichtet wird, dann wird sein Wert durch Kopie gebildet. Das heißt im Beispiel, my-const-1 und my-const-2 sind zwei verschiedene Entitäten, die wertgleich sind.

(Idealisiertes) Objektmodell

TECHNISCHE UNIVERSITÄT DARMSTADT

- Es gibt keine Objekte, nur Werte
  - Werte sind *immer*(?) Konstante, *nie*(?) Variable
  - Werte werden *immer* kopiert
- Laufzeitsystem kann intern zur Optimierung von dieser Grundlogik abweichen
  - Ist für Programmierer nicht sichtbar

```
( define my-const-1 12345 )
( define  my-const-2 my-const-1 )
( define ( my-fct my-const ) ( ..... ) )
```

Das gilt genauso für Parameter: Der formale Parameter innerhalb der Funktion ist eine Kopie des aktualen Parameters.

(Idealisiertes) Objektmodell

TECHNISCHE UNIVERSITÄT DARMSTADT

- Es gibt keine Objekte, nur Werte
  - Werte sind *immer*(?) Konstante, *nie*(?) Variable
  - Werte werden *immer* kopiert
- Laufzeitsystem kann intern zur Optimierung von dieser Grundlogik abweichen
  - Ist für Programmierer nicht sichtbar

```
( define my-const-1 12345 )
( define  my-const-2 my-const-1 )
( define ( my-fct my-const ) ( ..... ) )
```

Die oben formulierten Grundprinzipien des Objektmodells sind aber nur das, was nach außen hin, also für Racket-Programmierer sichtbar ist. Gerade deklarative Sprachen bieten dem Laufzeitsystem viel Potential, die Ausführung eines Programms intern und somit unsichtbar zu optimieren.

(Idealisiertes) Objektmodell

TECHNISCHE UNIVERSITÄT DARMSTADT

- Es gibt keine Objekte, nur Werte
  - Werte sind *immer*(?) Konstante, *nie*(?) Variable
  - Werte werden *immer* kopiert
- Laufzeitsystem kann intern zur Optimierung von dieser Grundlogik abweichen
  - Ist für Programmierer nicht sichtbar

```
( define my-const-1 12345 )
( define  my-const-2 my-const-1 )
( define ( my-fct my-const ) ( ..... ) )
```

Die Fragezeichen greifen das Wort „Idealisiertes“ in Klammern auf: In Kapitel 04d, Abschnitt „Aufweichung der reinen funktionalen Lehre in Racket“, werden wir weitere Sprachkonstrukte von Racket sehen, die das hier vorgestellte Objektmodell dann doch etwas durchbrechen.

TECHNISCHE UNIVERSITÄT DARMSTADT

Definitionen verstecken

In Java gibt es ja bekanntlich die Möglichkeit, Definitionen zu verstecken, damit Hilfskonstrukte, die nur lokal Sinn machen, nicht nach außen sichtbar werden. Das betrifft einerseits Referenztypen, die in ihren Quelldateien nicht public definiert sind, andererseits Attribute und Methoden, die in ihren Klassen nicht public definiert sind.

Wir werden als nächstes sehen, was da in Racket so geht. Da in Racket Funktionen die grundlegenden Bausteine sind, wird es nicht überraschen, dass Funktionen versteckt werden, aber auch die Definition von Konstanten.

Definitionen „verstecken“ in Java

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public class {
    private int n;
    private void m1()  { ..... }
    public  void m2()  { ..... }
    public  void m3()  { ..... }
}
```

- Attribut n und Methode m1 sind nur in m1, m2 und m3 sichtbar

Hier sehen Sie noch einmal ein einfaches Beispiel für versteckte Attribute einer Klasse in Java.

Analog in Racket

TECHNISCHE UNIVERSITÄT DARMSTADT

```
;; fct: number -> number
( define ( fct x )
     ( local
       ( ( define const 10 )
         ( define ( mult-const y ) ( * const y ) ) )
       ( + const ( mult-const x ) ) ) )
```

- Konstante const und Funktion mult-const sind nur im local-Ausdruck sichtbar

Wie gesagt, in Racket sind ja nicht Klassen und Objekte, sondern Funktionen die zentralen Strukturierungselemente. Daher finden wir die Möglichkeit, Definitionen zu verstecken, eben bei Funktionen.

Analog in Racket

TECHNISCHE UNIVERSITÄT DARMSTADT

```
;; fct: number -> number
( define ( fct x )
      ( local
        ( ( define const 10 )
          ( define ( mult-const y ) ( * const y ) ) )
        ( + const ( mult-const x ) ) ) )
```

- Konstante const und Funktion mult-const sind nur im local-Ausdruck sichtbar

Wieder einmal keine Funktion, die einen sinnvollen Wert zurückliefert, sondern einfach nur ein illustratives Beispiel, um zu zeigen, wie der Mechanismus funktioniert.

Analog in Racket

TECHNISCHE UNIVERSITÄT DARMSTADT

```
;; fct: number -> number
( define ( fct x )
    ( local
      ( ( define const 10 )
        ( define ( mult-const y ) ( * const y ) ) )
      ( + const ( mult-const x ) ) ) )
```

- Konstante const und Funktion mult-const sind nur im local-Ausdruck sichtbar

Bisher hatten wir einfache Ausdrücke und zusammengesetzte Ausdrücke kennen gelernt. Jetzt sehen wir eine vierte Art, nämlich einen local-Ausdruck.

Analog in Racket

TECHNISCHE UNIVERSITÄT DARMSTADT

```
;; fct: number -> number
( define ( fct x )
     ( local
       ( ( define const 10 )
         ( define ( mult-const y ) ( * const y ) ) )
       ( + const ( mult-const x ) ) ) )
```

- Konstante const und Funktion mult-const sind nur im local-Ausdruck sichtbar

Ganz am Ende des local-Ausdrucks steht eine beliebige Art von Ausdruck. Meist ist dieser Ausdruck recht kurz wie hier, aber das muss nicht sein. Der Wert dieses Ausdrucks ist der Wert des gesamten local-Ausdrucks.

Analog in Racket

TECHNISCHE UNIVERSITÄT DARMSTADT

```
;; fct: number -> number
( define ( fct x )
    ( local
      ( ( define const 10 )
        ( define ( mult-const y ) ( * const y ) ) )
      ( + const ( mult-const x ) ) ) )
```

- Konstante const und Funktion mult-const sind nur im local-Ausdruck sichtbar

Zwischen dem Schlüsselwort local und diesem Ausdruck können beliebig viele Definitionen kommen. Diese Definitionen können innerhalb des local-Ausdrucks verwendet werden, sind aber außerhalb der Funktion fct nicht sichtbar.

Ein local-Ausdruck ist also einfach ein Ausdruck wie jeder andere auch, nur dass ihm eigene Definitionen vorangehen, alles zusammen eingeleitet durch Schlüsselwort local.

Analog in Racket

TECHNISCHE UNIVERSITÄT DARMSTADT

```
;; fct: number -> number
( define ( fct x )
     ( local
       ( ( define const 10 )
         ( define ( mult-const y ) ( * const y ) ) )
       ( + const ( mult-const x ) ) ) )
```

- Konstante const und Funktion mult-const sind nur im local-Ausdruck sichtbar

Allediese lokalen Definitionen werden zur Abgrenzung noch einmal in ein Extra-Klammerpaar eingefasst.

TECHNISCHE UNIVERSITÄT DARMSTADT

Rekursion in Java

Bevor wir mit Racket weitermachen, bleiben wir kurz in Java und betrachten ein fundamentales Programmierkonzept in eigentlich allen modernen Sprachen: Rekursion. In funktionalen Sprachen wie Racket ist Rekursion das grundlegende Konzept für alles andere. Aber auch in Java ist Rekursion häufig sehr nützlich, daher wird es sowieso langsam Zeit, Rekursion einmal zu behandeln.

Rekursion in Java

TECHNISCHE UNIVERSITÄT DARMSTADT

```
// Precondition: n >= 0
public static void m ( int n ) {
   System.out.println ( n );
   if ( n > 0 )
      m ( n – 1 );
   System.out.println ( n );
}
```

```
m ( 3 );

3
2
1
0
0
1
2
3
```

Als erstes schauen wir uns eine rein illustrative Klassenmethode an, die zeigt, was Rekursion ist und wie es funktioniert. Auf den weiteren Folien schauen wir uns dann sinnvolle Beispiele an.

Rekursion in Java

TECHNISCHE UNIVERSITÄT DARMSTADT

```
// Precondition: n >= 0
public static void m ( int n ) {
   System.out.println ( n );
   if ( n > 0 )
      m ( n – 1 );
   System.out.println ( n );
}
```

```
m ( 3 );

3
2
1
0
0
1
2
3
```

Hier sehen Sie, was Rekursion meint: Eine Methode ruft sich selbst auf. Die ganze Methode m besteht also nur darin, dass sie sich mit 1 weniger selbst aufruft und vor und nach diesem rekursiven Aufruf jeweils der aktuale Parameter ausgegeben wird.

Rekursion in Java

TECHNISCHE UNIVERSITÄT DARMSTADT

```
// Precondition: n >= 0
public static void m ( int n ) {
  System.out.println ( n );
  if ( n > 0 )
    m ( n – 1 );
  System.out.println ( n );
}
```

```
m ( 3 );

3
2
1
0
0
1
2
3
```

Wenn die Methode m mit aktualem Parameter 3 aufgerufen wird, dann wird m in diesem Aufruf mit aktualem Parameter 2 aufgerufen, aber vorher wird die 3 ausgegeben. In dem Aufruf mit aktualem Parameter 2 wird m mit aktualem Parameter 1 aufgerufen, aber vorher wird die 2 ausgegeben, und so weiter. Bei aktualem Parameter 0 bricht die Rekursion ab.

Rekursion in Java

TECHNISCHE UNIVERSITÄT DARMSTADT

```
// Precondition: n >= 0
public static void m ( int n ) {
   System.out.println ( n );
   if ( n > 0 )
      m ( n – 1 );
   System.out.println ( n );
}
```

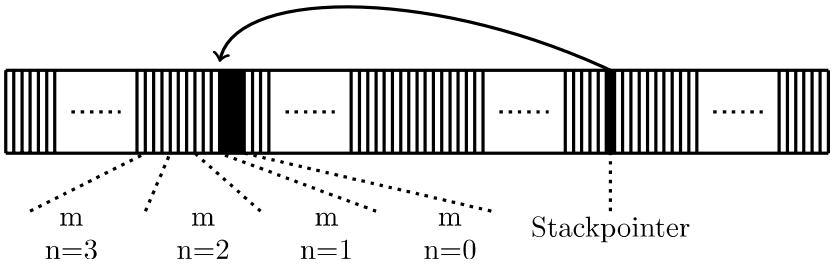
```
m ( 3 );

3
2
1
0
0
1
2
3
```

Bevor die Methode m mit aktualem Parameter 0 beendet wird, wird noch einmal die 0 ausgegeben. Dann geht der Prozess in den Aufruf von m mit aktualem Parameter 1 zurück, und es wird die 1 ausgegeben. Analog geht der Prozess dann zurück in den Aufruf mit 2 und dann in den mit 3. Jeder dieser Aufrufe gibt jeweils noch seinen aktualen Parameter aus, bevor er beendet wird.

Rekursion in Java

TECHNISCHE UNIVERSITÄT DARMSTADT



Der Blick auf den Call-Stack zeigt, was dahintersteckt: In jedem Frame, am selben Offset von der Startadresse des jeweiligen Frames, ist ein Platz für den Parameter n reserviert. Der ist bei jedem rekursiven Aufruf ein anderer, nämlich immer um 1 kleiner. In jeder Bildschirmausgabe wird immer der Wert von n im obersten Frame genommen, also in dem Frame, auf dessen Anfangsadresse der Stack-Pointer momentan verweist.

Rekursion in Java

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public static void m ( int n ) {
   System.out.println ( n );
   m ( n – 1 );
   System.out.println ( n );
}
```

```
m ( 3 );

   3
   2
   1
   0
 – 1
 – 2
 – 3
 .....
```

Noch eine kleine Variation des rein illustrativen ersten Beispiels: Der rekursive Abbruch bei aktualem Parameter 0 ist jetzt eliminiert, das heißt, bei jedem aktualen Parameter geht die Rekursion weiter. Rechts sehen Sie die Konsequenz: Die Methode m wird eins um andere Mal aufgerufen, aber keiner dieser Aufrufe wird je wieder beendet.

Das geht natürlich nicht beliebig lange gut. Bei jedem rekursiven Aufruf wird ja ein neuer Frame auf den Call-Stack gepackt. Irgendwann ist der für den Call-Stack reservierte Speicherplatz aufgebraucht, und das Laufzeitsystem bricht die Ausführung des Programms ab.

Rekursion in Java

TECHNISCHE UNIVERSITÄT DARMSTADT

$0! = 1$

$n! = n \cdot (n-1)!$ für $n > 0$

fib (0) = fib (1) = 1

fib (n) = fib ($n-1$) + fib ($n-2$) für $n > 1$

$\binom{n}{k} = 0$ für n < 0 oder k > n

$\binom{n}{k} = 1$ für n = 0 oder n = k

$\binom{n}{k} = \binom{n-1}{k} + \binom{n-1}{k-1}$

Wir kommen jetzt zu drei realen Beispielen aus der Mathematik. In der Mathematik gibt es für viele Funktionen auf den natürlichen Zahlen rekursive Definitionen, zum Beispiel für diese drei: die Fakultät, die Folge der Fibonacci-Zahlen und den Binomialkoeffizient. Auf den nächsten Folien werden wir diese drei Funktionen eins-zu-eins in rekursive Java-Methoden übertragen.

Rekursion in Java

TECHNISCHE UNIVERSITÄT DARMSTADT

```
// Precondition: n >= 0
public static int factorial ( int n ) {
   int result = 1;
   if ( n > 0 )
      result = n * factorial ( n – 1 );
   System.out.print ( n + " : " );
   System.out.println ( result );
   return result;
}
```

```
factorial ( 6 );

0 : 1
1 : 1
2 : 2
3 : 6
4 : 24
5 : 120
6 : 720
```

Das ist das erste Beispiel: die Fakultätsfunktion.

Rekursion in Java

TECHNISCHE UNIVERSITÄT DARMSTADT

```
// Precondition: n >= 0
public static int factorial ( int n ) {
   int result = 1;
   if ( n > 0 )
      result = n * factorial ( n – 1 );
   System.out.print ( n + " : " );
   System.out.println ( result );
   return result;
}
```

```
factorial ( 6 );

0 : 1
1 : 1
2 : 2
3 : 6
4 : 24
5 : 120
6 : 720
```

Speziell hier sehen Sie wie versprochen die Eins-zu-eins-Übersetzung der rekursiven Definition in rekursiven Java-Code.

Rekursion in Java

TECHNISCHE UNIVERSITÄT DARMSTADT

```
// Precondition: n >= 0
public static int factorial ( int n ) {
   int result = 1;
   if ( n > 0 )
      result = n * factorial ( n – 1 );
   System.out.print ( n + " : " );
   System.out.println ( result );
   return result;
}
```

```
factorial ( 6 );
```

```
0 : 1
1 : 1
2 : 2
3 : 6
4 : 24
5 : 120
6 : 720
```

Normalerweise sollte man in einer solchen Funktion natürlich keine Schreibausgaben haben. Hier haben wir sie nur zur Illustration der Methode.

Der Methodenaufruf mit aktualem Parameter 0 kommt als erster bis zu dieser Schreibausgabe. Daher wird als erstes der Wert 0 für n ausgegeben. Da der initiale Wert 1 für result bei n gleich 0 nicht durch einen rekursiven Aufruf von factorial überschrieben wird, bleibt es bei Wert 1 in result, und das ist ja auch korrekt.

Danach ist der Aufruf mit n gleich 0 zu Ende, und sein Rückgabewert wird mit n gleich 1 multipliziert. Danach ist der Aufruf mit n gleich 1 zu Ende, und sein Rückgabewert wird mit n gleich 2 multipliziert, und so weiter.

Rekursion in Java

TECHNISCHE UNIVERSITÄT DARMSTADT

```
// Precondition: n >= 0

public static int fibonacci ( int n ) {
   if ( n <= 1 )
      return 1;
   return fibonacci ( n – 1 ) + fibonacci ( n – 2 );
}
```

Analog sieht es beim zweiten Beispiel aus.

Rekursion in Java

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public static int binomialCoefficient ( int n, int k ) {
   if ( n < 0 || k > n )
      return 0;
   if ( n == 0 || n == k )
      return 1;
   return binomialCoefficient ( n – 1, k )
         + binomialCoefficient ( n – 1, k – 1 );
}
```

Es fehlt noch der Binomialkoeffizient aus der Eingangsliste von rekursiven mathematischen Definitionen. Die gesamte mathematische Definition des Binomialkoeffizienten besteht aus drei Fällen, und diese sind nacheinander eins-zu-eins in dieser Methode in Java umgesetzt. Der dritte Fall ist der rekursive.

Rekursion in Java

TECHNISCHE UNIVERSITÄT DARMSTADT

Nullstellenberechnung bei einer stetigen Funktion mit Vorzeichenwechsel:

- ***Beachte*: x * y <= 0 ist äquivalent dazu, dass entweder x >= 0 und y <= 0 oder x <= 0 und y >= 0 ist**
- **Sei Funktion f:$\mathbb{R}$→ $\mathbb{R}$ stetig im Intervall [a,b], sei f(a) * f(b) <= 0, dann gilt:**
  - **f hat mindestens eine Nullstelle in [a,b]**
  - ***Rekursion*: Sei m := (b – a) / 2; ist f(a) * f(m) <= 0, dann hat f in [a,m] mindestens eine Nullstelle, sonst mindestens eine Nullstelle in [m,b]**

Nun zu einem weiteren Beispiel für Rekursion, dem vierten. Das ist das aus der Mathematik bekannte Bisektionsverfahren zur Berechnung einer Nullstelle. Hier geht es nicht mehr um natürliche Zahlen wie in den vorangegangenen Beispielen, sondern um reelle Zahlen.

Rekursion in Java

TECHNISCHE UNIVERSITÄT DARMSTADT

Nullstellenberechnung bei einer stetigen Funktion mit Vorzeichenwechsel:

- ***Beachte***: x * y <= 0 ist äquivalent dazu, dass entweder x >= 0 und y <= 0 oder x <= 0 und y >= 0 ist
- Sei Funktion f:$\mathbb{R} \rightarrow \mathbb{R}$ stetig im Intervall [a,b], sei f(a) * f(b) <= 0, dann gilt:
  - f hat mindestens eine Nullstelle in [a,b]
  - ***Rekursion***: Sei m := (b – a) / 2; ist f(a) * f(m) <= 0, dann hat f in [a,m] mindestens eine Nullstelle, sonst mindestens eine Nullstelle in [m,b]

Den Vorzeichenwechsel werden wir durch diese offensichtliche Regel prüfen.

Rekursion in Java

TECHNISCHE UNIVERSITÄT DARMSTADT

Nullstellenberechnung bei einer stetigen Funktion mit Vorzeichenwechsel:

- ***Beachte***: x * y <= 0 ist äquivalent dazu, dass entweder x >= 0 und y <= 0 oder x <= 0 und y >= 0 ist
- Sei Funktion f:$\mathbb{R} \rightarrow \mathbb{R}$ stetig im Intervall [a,b], sei f(a) * f(b) <= 0, dann gilt:
  - f hat mindestens eine Nullstelle in [a,b]
  - ***Rekursion***: Sei m := (b – a) / 2; ist f(a) * f(m) <= 0, dann hat f in [a,m] mindestens eine Nullstelle, sonst mindestens eine Nullstelle in [m,b]

Das ist die zugrundeliegende mathematische Erkenntnis, die garantiert, dass die Funktion f tatsächlich mindestens eine Nullstelle im Intervall [a,b] hat.

Rekursion in Java

TECHNISCHE UNIVERSITÄT DARMSTADT

Nullstellenberechnung bei einer stetigen Funktion mit Vorzeichenwechsel:

- ***Beachte***: x * y <= 0 ist äquivalent dazu, dass entweder x >= 0 und y <= 0 oder x <= 0 und y >= 0 ist
- Sei Funktion f:$\mathbb{R} \rightarrow \mathbb{R}$ stetig im Intervall [a,b], sei f(a) * f(b) <= 0, dann gilt:
  - f hat mindestens eine Nullstelle in [a,b]
  - ***Rekursion***: Sei m := (b – a) / 2; ist f(a) * f(m) <= 0, dann hat f in [a,m] mindestens eine Nullstelle, sonst mindestens eine Nullstelle in [m,b]

Die hier farblich unterlegte Aussage ist sozusagen die Quintessenz des Bisektionsverfahrens: Wenn im Intervall [a,b] mindestens eine Nullstelle ist, dann gilt dies auch für das Intervall [a,m] oder für das Intervall [m,b] oder sogar für beide. Mindestens eines dieser beiden Teilintervalle hat einen Vorzeichenwechsel. Auf dieses Teilintervall können wir uns konzentrieren und die andere Hälfte des ursprünglichen Intervalls unbesehen vergessen.

Rekursion in Java

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public interface DoubleToDoubleFunction {
   public double apply ( double x );
}

public class QuadraticFunction implements DoubleToDoubleFunction {
   private double summand;
   private double factor:
   public QuadraticFunction ( double summand, double factor ) {
      this.summand = summand;
      this.factor = factor;
   }
   public double apply ( double x ) {
       return x * x * factor + summand;
   }
}
```

Für eine möglichst allgemeine Implementation dieser Rekursion erinnern wir uns noch einmal an ein Interface und eine Klasse aus Kapitel 03b, Abschnitt zu Interfaces: Das Interface oben repräsentiert das allgemeine Konzept „reellwertige Funktion mit reellwertigem Argument“. Implementierende Klassen repräsentieren dann konkrete Funktionen, so wie die Klasse unten, die allgemeine quadratische Funktionen realisiert. Wir haben in jenem Abschnitt aber auch implementierende Klassen gesehen, die statt dessen die Summe oder auch die Komposition zweier Funktionen realisieren, die ihrerseits wieder durch das Interface oben gegeben sind. Dieses Design für mathematische Funktionen ist also sehr flexibel verwendbar, um beliebige und beliebig komplexe Funktionen aufzubauen.

Rekursion in Java

TECHNISCHE UNIVERSITÄT DARMSTADT

```
// Precondition: a < b, f.apply(a) * f.apply(b) <= 0, epsilon > 0, and
//         the mathematical function represented by f is continuous
public static double findZero ( double a, double b, double epsilon,
                                DoubleToDoubleFunction f ) {
   double m = ( a + b ) / 2;
   if ( m – a < epsilon )
      return m;
   if ( f.apply(a) * f.apply(m) >= 0 )
      return findZero ( m, b, epsilon, f );
   return findZero ( a, m, epsilon, f );
}
```

Dies ist nun unsere rekursive Implementation des Bisektionsverfahrens in Java.

Rekursion in Java

TECHNISCHE UNIVERSITÄT DARMSTADT

```
// Precondition: a < b, f.apply(a) * f.apply(b) <= 0, epsilon > 0, and
//         the mathematical function represented by f is continuous
public static double findZero ( double a, double b, double epsilon,
                                DoubleToDoubleFunction f ) {
   double m = ( a + b ) / 2;
   if ( m – a < epsilon )
      return m;
   if ( f.apply(a) * f.apply(m) >= 0 )
      return findZero ( m, b, epsilon, f );
   return findZero ( a, m, epsilon, f );
}
```

Hier finden Sie als Kommentar nochmals die Vorbedingungen, die von der zu untersuchenden Funktion erfüllt sein müssen, damit das Bisektionsverfahren garantiert eine Nullstelle findet.

Rekursion in Java

TECHNISCHE UNIVERSITÄT DARMSTADT

```
// Precondition: a < b, f.apply(a) * f.apply(b) <= 0, epsilon > 0, and
//         the mathematical function represented by f is continuous
public static double findZero ( double a, double b, double epsilon,
                                DoubleToDoubleFunction f ) {
   double m = ( a + b ) / 2;
   if ( m – a < epsilon )
      return m;
   if ( f.apply(a) * f.apply(m) >= 0 )
      return findZero ( m, b, epsilon, f );
   return findZero ( a, m, epsilon, f );
}
```

Das Bisektionsverfahren grenzt eine Nullstelle immer weiter ein. Da Nullstellen potentiell irrational sein können, lässt sich die eingegrenzte Nullstelle in der Regel nicht exakt bestimmen. Wir lösen dieses Problem so, dass wir mittels Parameter eine Maximaldistanz zwischen der Mitte und den beiden Enden des Intervalls vorgeben, und wenn diese Maximaldistanz unterschritten ist, sind wir zufrieden. Dies ist eine gängige Vorgehensweise.

***Nebenbemerkung*: Hier geben wir der Funktion findZero eine *absolute* Fehlertoleranz vor. Man hätte auch eine *relative* Fehlertoleranz vorgeben können nämlich relativ zur Intervallgröße, also beispielsweise ein Millionstel der Intervallgröße. Es hängt von den konkreten Umständen ab, ob eine absolute oder eine relative Fehlertoleranz zielführend ist, aber das ist nicht mehr Thema der FOP, sondern von Lehrveranstaltungen zur numerischen Mathematik.**

Rekursion in Java

TECHNISCHE UNIVERSITÄT DARMSTADT

```
// Precondition: a < b, f.apply(a) * f.apply(b) <= 0, epsilon > 0, and
//         the mathematical function represented by f is continuous
public static double findZero ( double a, double b, double epsilon,
                                DoubleToDoubleFunction f ) {
   double m = ( a + b ) / 2;
   if ( m – a < epsilon )
      return m;
   if ( f.apply(a) * f.apply(m) >= 0 )
      return findZero ( m, b, epsilon, f );
   return findZero ( a, m, epsilon, f );
}
```

Wenn das Intervall noch zu groß ist, halbieren wir es, indem wir entweder die linke oder die rechte Seite durch die Mitte ersetzen. Aber dabei müssen wir darauf achten, dass im halbierten Intervall auf jeden Fall noch eine Nullstelle ist. Wie wir schon diskutiert hatten, ist das bei einer stetigen Funktion auf jeden Fall bei Vorzeichenwechsel so.

Rekursion in Java

TECHNISCHE UNIVERSITÄT DARMSTADT

```
// Precondition: a < b, f.apply(a) * f.apply(b) <= 0, epsilon > 0, and
//         the mathematical function represented by f is continuous
public static double findZero ( double a, double b, double epsilon,
                                DoubleToDoubleFunction f ) {
   double m = ( a + b ) / 2;
   if ( m – a < epsilon )
      return m;
   if ( f.apply(a) * f.apply(m) >= 0 )
      return findZero ( m, b, epsilon, f );
   return findZero ( a, m, epsilon, f );
}
```

Hier ist jetzt die Rekursion: Falls das halbierte Intervall vom linken Ende bis zur Mitte des bisherigen Intervalls sicher eine Nullstelle enthält, dann macht der Algorithmus rekursiv mit diesem Intervall weiter. Ansonsten wissen wir nicht, ob dieses Intervall eine Nullstelle enthält, aber wir wissen dann, dass das halbierte Intervall von der Mitte bis zum rechten Ende eine Nullstelle enthält, daher macht der Algorithmus im zweiten Fall mit dem letzteren Intervall weiter.

TECHNISCHE UNIVERSITÄT DARMSTADT

Rekursion in Racket

Wir haben nun Rekursion in Java gesehen und vorher die Grundlagen von Racket. In Racket ist Rekursion das grundlegende Konzept zur Steuerung des Programmablaufs in einer Funktion, ähnlich zu Schleifen in Java.

***Vorgriff*: In Kapitel 04d werden wir sehen, dass Schleifen eigentlich dem Geist der funktionalen Programmierung widersprechen und in funktionalen Sprachen wie Racket auch nicht wirklich realisiert sind, allenfalls simuliert werden können durch Sprachkonstrukte, die dem Geist der funktionalen Programmierung widersprechen.**

Rekursive Funktionen

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public class Class4 {
    public static int factorial ( int n ) {
        return n == 0 ? 1 : n * factorial ( n – 1 );
}
```

```
;; Type: natural -> natural
( define ( factorial n )
      (  if  ( = n 0 )  1  ( * n ( factorial ( – n 1 ) ) )  )   )
```

Alle bis jetzt in Racket betrachteten Funktionen waren sehr einfach strukturiert. Nun sehen wir uns eine erste *rekursive* Funktion an.

Rekursive Funktionen

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public class Class4 {
    public static int factorial ( int n ) {
        return n == 0 ? 1 : n * factorial ( n – 1 );
}
```

```
;; Type: natural -> natural
( define ( factorial n )
      (  if  ( = n 0 )  1  ( * n ( factorial ( – n 1 ) ) )  )   )
```

Dieses Beispiel hatten wir schon weiter vorne in diesem Kapitel gesehen, bei Rekursion in Java. Zur besseren Gegenüberstellung mit Racket verwenden wir hier den Bedingungsoperator anstelle der if-Verzweigung.

Rekursive Funktionen

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public class Class4 {
    public static int factorial ( int n ) {
        return n == 0 ? 1 : n * factorial ( n – 1 );
}
```

```
;; Type: natural -> natural
( define ( factorial n )
      (  if  ( = n 0 )  1  ( * n ( factorial ( – n 1 ) ) )  )   )
```

Die Fakultätsfunktion ist auf den natürlichen Zahlen definiert und liefert eine natürliche Zahl zurück.

Rekursive Funktionen

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public class Class4 {
    public static int factorial ( int n ) {
        return n == 0 ? 1 : n * factorial ( n – 1 );
}


;; Type: natural -> natural
( define ( factorial n )
      (  if  ( = n 0 )  1  ( * n ( factorial ( – n 1 ) ) )  )   )
```

Soweit noch nichts Neues: Schlüsselwort define gefolgt von dem Namen der Funktion zusammen mit der einelementigen Parameterliste, wobei wie immer der Name der Funktion und der Name des Parameters in Klammern zusammengefasst sein müssen.

Rekursive Funktionen

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public class Class4 {
    public static int factorial ( int n ) {
        return n == 0 ? 1 : n * factorial ( n – 1 );
}


;; Type: natural -> natural
( define ( factorial n )
      (  if  ( = n 0 )  1  ( * n ( factorial ( – n 1 ) ) )  )   )
```

Die if-Verzweigung in Racket haben wir weiter vorne in diesem Kapitel gesehen.

Rekursive Funktionen

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public class Class4 {
    public static int factorial ( int n ) {
        return n == 0 ? 1 : n * factorial ( n – 1 );
}


;; Type: natural -> natural
( define ( factorial n )
      (  if  ( = n 0 )  1  ( * n ( factorial ( – n 1 ) ) )  )   )
```

Wie wir dort gesehen hatten, geht die Analogie zwischen Bedingungsoperator in Java und if-Abfrage in Racket sehr weit. Beide haben drei Ausdrücke als Operanden, und der erste Ausdruck ist boolesch.

Rekursive Funktionen

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public class Class4 {
    public static int factorial ( int n ) {
        return n == 0 ? 1 : n * factorial ( n – 1 );
}


;; Type: natural -> natural
( define ( factorial n )
      (  if  ( = n 0 )  1  ( * n ( factorial ( – n 1 ) ) )  )   )
```

Der zweite Operand liefert den Wert des Gesamtausdrucks für den Fall, dass der erste Operand true ergibt.

Rekursive Funktionen

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public class Class4 {
    public static int factorial ( int n ) {
        return n == 0 ? 1 : n * factorial ( n – 1 );
}


;; Type: natural -> natural
( define ( factorial n )
      (  if  ( = n 0 )  1  ( * n ( factorial ( – n 1 ) ) )  )   )
```

Und der dritte Operand liefert den Wert des Gesamtausdrucks im Fall, dass der erste Operand false ergibt.

Rekursive Funktionen

TECHNISCHE UNIVERSITÄT DARMSTADT

```
( factorial 4 )
( * 4 ( factorial 3 ) )
( * 4 ( * 3 ( factorial 2 ) ) )
( * 4 ( * 3 ( * 2 ( factorial 1 ) ) ) )
( * 4 ( * 3 ( * 2 ( * 1 ( factorial 0 ) ) ) ) )
( * 4 ( * 3 ( * 2 ( * 1 1 ) ) ) )
( * 4 ( * 3 ( * 2 1 ) ) )
( * 4 ( * 3 2 ) )
( * 4 6 )
```

Für Parameterwert 4 schauen wir uns einmal an, wie das Ergebnis schrittweise von DrRacket berechnet wird.

Rekursive Funktionen

TECHNISCHE UNIVERSITÄT DARMSTADT

```
( factorial 4 )
( * 4 ( factorial 3 ) )
( * 4 ( * 3 ( factorial 2 ) ) )
( * 4 ( * 3 ( * 2 ( factorial 1 ) ) ) )
( * 4 ( * 3 ( * 2 ( * 1 ( factorial 0 ) ) ) ) )
( * 4 ( * 3 ( * 2 ( * 1 1 ) ) ) )
( * 4 ( * 3 ( * 2 1 ) ) )
( * 4 ( * 3 2 ) )
( * 4 6 )
```

Da 4 ungleich 0 ist, ist die Fakultät von 4 nicht gleich 1, sondern gleich der Fakultät von 3 multipliziert mit 4.

Rekursive Funktionen

TECHNISCHE UNIVERSITÄT DARMSTADT

```
( factorial 4 )
( * 4 ( factorial 3 ) )
( * 4 ( * 3 ( factorial 2 ) ) )
( * 4 ( * 3 ( * 2 ( factorial 1 ) ) ) )
( * 4 ( * 3 ( * 2 ( * 1 ( factorial 0 ) ) ) ) )
( * 4 ( * 3 ( * 2 ( * 1 1 ) ) ) )
( * 4 ( * 3 ( * 2 1 ) ) )
( * 4 ( * 3 2 ) )
( * 4 6 )
```

Genau dasselbe passiert beim Parameterwert 3.

Rekursive Funktionen

TECHNISCHE UNIVERSITÄT DARMSTADT

```
( factorial 4 )
( * 4 ( factorial 3 ) )
( * 4 ( * 3 ( factorial 2 ) ) )
( * 4 ( * 3 ( * 2 ( factorial 1 ) ) ) )
( * 4 ( * 3 ( * 2 ( * 1 ( factorial 0 ) ) ) ) )
( * 4 ( * 3 ( * 2 ( * 1 1 ) ) ) )
( * 4 ( * 3 ( * 2 1 ) ) )
( * 4 ( * 3 2 ) )
( * 4 6 )
```

Und beim Parameterwert 2.

Rekursive Funktionen

TECHNISCHE UNIVERSITÄT DARMSTADT

```
( factorial 4 )
( * 4 ( factorial 3 ) )
( * 4 ( * 3 ( factorial 2 ) ) )
( * 4 ( * 3 ( * 2 ( factorial 1 ) ) ) )
( * 4 ( * 3 ( * 2 ( * 1 ( factorial 0 ) ) ) ) )
( * 4 ( * 3 ( * 2 ( * 1 1 ) ) ) )
( * 4 ( * 3 ( * 2 1 ) ) )
( * 4 ( * 3 2 ) )
( * 4 6 )
```

Und auch noch einmal beim Parameterwert 1.

Rekursive Funktionen

TECHNISCHE UNIVERSITÄT DARMSTADT

```
( factorial 4 )
( * 4 ( factorial 3 ) )
( * 4 ( * 3 ( factorial 2 ) ) )
( * 4 ( * 3 ( * 2 ( factorial 1 ) ) ) )
( * 4 ( * 3 ( * 2 ( * 1 ( factorial 0 ) ) ) ) )
( * 4 ( * 3 ( * 2 ( * 1 1 ) ) ) )
( * 4 ( * 3 ( * 2 1 ) ) )
( * 4 ( * 3 2 ) )
( * 4 6 )
```

Im Fall 0 hingegen wird der Wert 1 eingesetzt. Ab jetzt ist das Ganze ein ganz normaler arithmetischer Ausdruck, der im Weiteren dann schrittweise ausgewertet wird, wie wir es zu Beginn schon bei arithmetischen Ausdrücken gesehen haben.

Rekursive Funktionen

TECHNISCHE UNIVERSITÄT DARMSTADT

```
;; Type: natural -> natural
;; Returns: the factorial n! of n, that is,
;; 1 if n = 0 and n * (n-1)! otherwise
( define ( factorial n )
    ( if ( = 0 n ) 1 ( * n ( factorial (– n 1 ) ) ) ) )
( check-expect ( factorial 0 ) 1 )
( check-expect ( factorial 5 ) 120 )
```

Wir sollten immer auch Checks zur Prüfung der Korrektheit einer Funktion einsetzen. Wichtig dabei ist, dass wir immer auch die Randfälle mit abprüfen. Der Randfall hier ist der Fall, dass der Parameter den Wert 0 hat.

Rekursive Funktionen

TECHNISCHE UNIVERSITÄT DARMSTADT

```
;; Type: natural -> natural
( define ( fibonacci n )
   (  if  ( or ( = n 0 ) ( = n 1 ) )
          1
          ( + ( fibonacci ( – n 1 ) ) ( fibonacci ( – n 2 ) ) ) ) )


check-expect ( ( fibonacci 0 ) 1 )
check-expect ( ( fibonacci 1 ) 1 )
check-expect ( ( fibonacci 20 ) 10946 )
```

Genauso übersetzt sich auch die Fibonacci-Folge eins-zu-eins in eine rekursive Funktion in Racket. Auch hier schreiben wir gleich ein paar Checks dazu, darunter auch die beiden Randfälle 0 und 1.

Rekursive Funktionen

TECHNISCHE UNIVERSITÄT DARMSTADT

```
;; Type: integer integer -> natural
( define ( binom-coeff n k )
   ( if ( or ( < n 0 ) ( < n k ) )
     0
     ( if ( or ( = n 0 ) ( = n k ) )
        1
        ( + ( binom-coeff ( – n 1 ) k )
             ( binom-coeff ( – n 1 ) ( – k 1 ) ) ) ) ) ) )

( check-expect ( binom-coeff 123 –345 ) 0 )
( check-expect ( binom-coeff –456 678 ) 0 )
( check-expect ( binom-coeff 789 0 ) 1 )
.....
```

Auch die Definition des Binomialkoeffizienten lässt sich eins-zu-eins übertragen.

Rekursive Funktionen

TECHNISCHE UNIVERSITÄT DARMSTADT

```
;; Type: integer integer -> natural
( define ( binom-coeff n k )
   ( if ( or ( < n 0 ) ( < n k ) )
      0
      ( if ( or ( = n 0 ) ( = n k ) )
         1
         ( + ( binom-coeff ( – n 1 ) k )
              ( binom-coeff ( – n 1 ) ( – k 1 ) ) ) ) ) ) )

( check-expect ( binom-coeff 123 –345 ) 0 )
( check-expect ( binom-coeff –456 678 ) 0 )
( check-expect ( binom-coeff 789 0 ) 1 )
.....
```

Da wir drei verschiedene Fälle haben, müssen wir zwei if-Verzweigungen ineinander schachteln.

Rekursive Funktionen

TECHNISCHE UNIVERSITÄT DARMSTADT

```
;; Type: integer integer -> natural
( define ( binom-coeff n k )
   ( if ( or ( < n 0 ) ( < n k ) )
     0
     ( if ( or ( = n 0 ) ( = n k ) )
        1
        ( + ( binom-coeff ( – n 1 ) k )
             ( binom-coeff ( – n 1 ) ( – k 1 ) ) ) ) ) ) )
```

```
( check-expect ( binom-coeff 123 –345 ) 0 )
( check-expect ( binom-coeff –456 678 ) 0 )
( check-expect ( binom-coeff 789 0 ) 1 )
.....
```

Auch diese Funktion sollten wir selbstverständlich auf Herz und Nieren testen. Das wird hier nur beispielhaft angedeutet, weitere Testfälle bleiben Ihrer konstruktiven Kreativität überlassen.

Rekursive Funktionen

TECHNISCHE UNIVERSITÄT DARMSTADT

```
( define epsilon 0.00001 )

;; Type: ( real -> real ) real real -> real
;; Precondition:
;;      a < b;
;;      f is a continuous function in the interval [ a ... b ];
;;      it is f(a) * f(b) <= 0.
;; Returns: a value in the interval [ a ... b ] that differs from
;; some zero of f by no more than epsilon.

( define ( find-zero f a b ) ( ..... ) )
```

Zum Abschluss schauen wir uns Methode findZero auch in Racket an. Hier ist das natürlich eine Funktion, keine Methode, und heißt find-zero nach Racket-Konvention.

Rekursive Funktionen

TECHNISCHE UNIVERSITÄT DARMSTADT

```
( define epsilon 0.00001 )

;; Type: ( real -> real ) real real -> real
;; Precondition:
;;     a < b;
;;     f is a continuous function in the interval [ a ... b ];
;;     it is f(a) * f(b) <= 0.
;; Returns: a value in the interval [ a ... b ] that differs from
;; some zero of f by no more than epsilon.

( define ( find-zero f a b ) ( ..... ) )
```

***Erinnerung*: Es soll ein Wert aus einem reellwertigen Intervall berechnet werden, dessen untere Grenze a und dessen obere Grenze b ist. Vorbedingung muss natürlich sein, dass a nicht größer als b ist.**

Rekursive Funktionen

TECHNISCHE UNIVERSITÄT DARMSTADT

```
( define epsilon 0.00001 )

;; Type: ( real -> real ) real real -> real
;; Precondition:
;;      a < b;
;;      f is a continuous function in the interval [ a ... b ];
;;      it is f(a) * f(b) <= 0.
;; Returns: a value in the interval [ a ... b ] that differs from
;; some zero of f by no more than epsilon.

( define ( find-zero f a b ) ( ..... ) )
```

Weiter Erinnerung: Die Funktion f muss in diesem Intervall stetig sein, und die zu definierende Funktion find-zero soll eine Nullstelle von f finden.

Rekursive Funktionen

TECHNISCHE UNIVERSITÄT DARMSTADT

```
( define epsilon 0.00001 )

;; Type: ( real -> real ) real real -> real
;; Precondition:
;;      a < b;
;;      f is a continuous function in the interval [ a ... b ];
;;      it is f(a) * f(b) <= 0.
;; Returns: a value in the interval [ a ... b ] that differs from
;; some zero of f by no more than epsilon.

( define ( find-zero f a b ) ( ..... ) )
```

Weiter Erinnerung: Aufgrund dieser Bedingung hat f einen Vorzeichenwechsel und daher garantiert mindestens eine Nullstelle im geschlossenen Intervall von a nach b.

Rekursive Funktionen

TECHNISCHE UNIVERSITÄT DARMSTADT

```
( define epsilon 0.00001 )

;; Type: ( real -> real ) real real -> real
;; Precondition:
;;      a < b;
;;      f is a continuous function in the interval [ a ... b ];
;;      it is f(a) * f(b) <= 0.
;; Returns: a value in the interval [ a ... b ] that differs from
;; some zero of f by no more than epsilon.

( define ( find-zero f a b ) ( ..... ) )
```

Weiter Erinnerung: Wir können die Nullstelle im Allgemeinen nicht exakt darstellen, sondern nur bis auf ein epsilon eingrenzen.

Rekursive Funktionen

TECHNISCHE UNIVERSITÄT DARMSTADT

```
( define ( find-zero f a b )
   ( local
     ( ( define m ( / ( + a b ) 2 ) ) )
     ( if ( < ( − b a ) epsilon )
          m
          ( if  ( > ( * ( f a ) ( f m ) ) 0 )
                ( find-zero f m b )
                ( find-zero f a m ) ) ) ) )
```

Nachdem wir den Vertrag der Funktion find-zero geklärt haben, können wir uns nun an die Implementation machen.

Rekursive Funktionen

TECHNISCHE UNIVERSITÄT DARMSTADT

```
( define ( find-zero f a b )
   ( local
     ( ( define m ( / ( + a b ) 2 ) ) )
     ( if ( < ( − b a ) epsilon )
          m
          ( if  ( > ( * ( f a ) ( f m ) ) 0 )
                ( find-zero f m b )
                ( find-zero f a m ) ) ) ) )
```

Die Berechnungsvorschrift ist genau dieselbe wie bei findZero in Java weiter vorne in diesem Kapitel.

Rekursive Funktionen

TECHNISCHE UNIVERSITÄT DARMSTADT

```
( define ( find-zero f a b )
   ( local
     ( ( define m ( / ( + a b ) 2 ) ) )
     ( if ( < ( − b a ) epsilon )
          m
          ( if  ( > ( * ( f a ) ( f m ) ) 0 )
                ( find-zero f m b )
                ( find-zero f a m ) ) ) ) )
```

Mit local definieren wir m intern, so wie wir das mit den Möglichkeiten von Java auch bei findZero gemacht haben.

Rekursive Funktionen

TECHNISCHE UNIVERSITÄT DARMSTADT

```
( define ( find-zero f a b )
   ( local
    ( ( define m ( / ( + a b ) 2 ) ) )
    ( if ( < ( − b a ) epsilon )
         m
         ( if  ( > ( * ( f a ) ( f m ) ) 0 )
               ( find-zero f m b )
               ( find-zero f a m ) ) ) ) )
```

***Erinnerung*: Alle lokalen Definitionen – in diesem Fall nur eine – werden noch einmal in Klammern zusammengefasst.**