Funktionen als Daten

Sven Eric Panitz

13. Januar 2023

Inhaltsverzeichnis

In diesem Lehrbrief geht es um Funktionen als Daten:

Funktionsdaten.java

```
package name.panitz.oose;
import java.util.function.*;
import java.util.*;
import java.time.*;
import java.util.concurrent.*;
```

```
public interface Funktionsdaten{
```

Wir werden im Zuge dieses Lehrbriefes eine Reihe von Funktionen höherer Ordnung schreiben, die auf Listen von Objekzten agieren. Daher geben wir hier noch einmal die Implementierung einer Liste, die auf einer Reihung als Datenspeicher basiert.

Funktionsdaten.java

```
class AL<E>{
  private int theSize = 0;
  private Object[] store = new Object[10];

  public int size(){return theSize;}

  @SuppressWarnings("unchecked")
  public E get(int i){
    if (i>=size()||i<0) throw new IndexOutOfBoundsException();
    return (E)store[i];
  }
  public void add(E e){
    if (theSize>=store.length) enlargeStore();
    store[theSize++] = e;
  }
  private void enlargeStore(){
    Object[] newStore = new Object[store.length+10];
    for (int i=0;i<theSize;i++) newStore[i]=store[i];
    store=newStore;
  }
  @Override public String toString(){
    var result = new StringBuffer("[");
    for (var i=0;i<size();i++){
      if (i>0) result.append(", ");
      result.append(store[i]);
    }
    result.append("]");
    return result.toString();
  }
}
```

Wir sehen auch wieder eine einfache Methode vor, um Listen durch Aufzählen der Elemente zu erzeugen.

Funktionsdaten.java

```
static <E> AL<E> of(E... es){
  var rs = new AL<E>();
  for (var e:es) rs.add(e);
  return rs;
}
```

Die Bezeichnung Lambda-Ausdruck geht zurück auf ein Berechenbarkeitskalkül, das der amerikanische Mathematiker Alonzo Church in der 30er Jahren entwickelt hat.[Chu36] Dort wird eine namenlose Funktionen mit dem griechischen Buchstaben Lambda λ eingeleitet, z.B. $\lambda x.2 * x$.
Zur gleichen Zeit entwickelte in England Alan Turing ein komplett anderes Berechenbarkeitsmodell, das wir heute als Turing Machine bezeichnen.[Tur36]
Während der Lambda-Kalkül ein theoretisches Modell der funktionalen Programmierung ist, abstrahiert die Turing Maschine heutige Rechenmaschinen.
Turing hat dann zeigen können, dass ihre beiden sehr unterschiedlichen Ansätze die gleiche Menge berechenbarer Funktionen definieren.[Tur37]

1 Funktionsvariablen

Es ist in Java auch möglich, eine Funktion in einer Variablen zu speichern. Der Typ der Variable muss hierfür als `Function` markiert sein. Zusätzlich ist an dem Typ `Function` zu notieren, von welchem Typ das Argument und von welchem Typ das Ergebnis der Funktion ist. Dieses wird in einem spitzen Klammernpaar dem Typ `Function` nachgestellt. Soll eine Funktion mit einem ganzzahligen Argument und einer ganzen Zahl als Ergebnis gespeichert werden, so beschreibt das der Typ `Function<Integer,Integer>`.

Es ist dabei zu beachten, dass Argumenttyp und Ergbnisbtyp einer Funktionsvariablen nur Referenztypen sein dürfen.

Ein Typ `Function<int,int>`, wie man eigentlich vermutet hätte, ist nicht erlaubt. Stattdessen ist die Klasse `Integer` zu verwenden.

Es gibt nun eine Kurznotation, um eine Funktion zu schreiben. Die Argumente der Funktion werden dabei in runden Klammern geschrieben. Es folgt ein aus einem Minus- und Größerzeichen stilisierter Pfeil, der von dem Ausdruck für das Ergebnis gefolgt wird.

Die Funktion, die ein Argument mit 2 multipliziert lässt sich also schreiben als: `(x) -> 2*x`.

Insgesamt können wir also folgende Funktionsvariable anlegen:

```
Function<Integer,Integer> f1 = (x) -> 2*x
```

Die Pfeilnotation als Kurznotation für eine Funktion, wird als Lambda-Ausdruck bezeichnet.

Des Wesen eines Lambda-Ausdrucks ist, dass eine Funktion definiert wird, ohne ihr einen spezifischen Namen zu geben. Man spricht auch von anonymen Funktionen.

Eine der ersten Programmiersprachen, die Lambda-Ausdrücke aufgenommen hat, war Lisp. Heutzutage haben fast alle höheren Programmiersprachen, in irgendeiner Weise Lambda-Ausdrücke integriert. In Java haben sie mit Version 1.8 Einzug genommen.

Eine Funktionsvariable enthält die Funktion, die aufzurufen ist. Hierzu verwendet man den Aufruf mit `apply`.

```
f1.apply(17+4)
```

```
$2 ==> 42
```

2 Funktionsargumente

In gleicher Weise, in der nun Funktionen als Variablen gespeichert werden können, können sie auch anderen Funktionen als Argument übergeben werden. Man spricht dann auch von Funktionen höherer Ordnung.

Eine Funktion höherer Ordnung ist folgende Funktion `twice`. Sie bekommt eine Funktion als Argument, sowie einen Parameter für diese Funktion. Die übergebene Funktion wird auf den Parameter angewendet und anschließend noch einmal auf das Ergebnis dieser Anwendung. Sie wird also zweimal hintereinander angewendet:

Funktionsdaten.java

```
static int twice1(Function<Integer,Integer> f, int i){
  return f.apply(f.apply(i));
}
```

Die Funktion `f1` zum Verdoppeln zweimal angewendet, vervierfacht also entsprechend das Argument:

```
twice1(f1,5)
```

```
$4 ==> 20
```

Wir können die Funktion nicht nur als Funktionsvariable, sondern auch direkt als Lambda-Ausdruck übergeben.

```
twice1((x) -> x*x,2)
```

```
$3 ==> 16
```

Beispiel 2.2 Funktionsargumente können hilfreich sein, wenn für eine Sammlung von Elementen eine Funktion auf alle Elemente dieser Sammlung anzuwenden ist.

Folgende Funktion erzeugt für eine Liste von Zahlen eine neue ebenso lange Liste Zahlen, indem eine Funktion der Reihe nach auf jedes Listenelement angewendet wird.

Funktionsdaten.java

```
static AL<Integer> aufAlle(AL<Integer> xs, Function<Integer,Integer> f){
  var rs = new AL<Integer>();
  for (var i=0; i<xs.size();i++)rs.add(f.apply(xs.get(i)));
  return rs;
}
```

Auf unsere Beispielliste der Zahlen 1 bis 10 angewendet können wir so zum Beispiel eine Liste von Quadratzahlen erhalten.

```
aufAlle(xs, (x) -> x*x);
```

```
$41 ==> [1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
```

3 Generische Funktionen höherer Ordnung

Die eben gesehene Funktion `twice1`, die eine Funktion doppelt anwendet, ist inhaltlich nicht auf den Typ von ganzen Zahlen beschränkt. Jede einstellige Funktion, deren Argument- und Ergebistyp gleich sind, kann mehrfach angewendet werden.

Wir haben schon gesehen, dass wir Funktionen über einen Typ abstrahieren können. Das lässt sich auch mit Funktionen höherer Ordnung machen. Wir können die Funktion `twice` verallgemeinern, sodass sie für einen beliebigen aber festen Typ `A`, eine Funktion mit Argument- und Ergebnistyp `A` doppelt anwendet.

Funktionsdaten.java

```
static <A> A twice(Function<A,A> f, A i){
  return f.apply(f.apply(i));
}
```

Diese allgemeinere Version lässt sich weiterhin auf eine Funktion, die mit ganzen Zahlen rechnet, anwenden.

```
twice((x)->x*x,5)
```

```
$68 ==> 625
```

Aber jetzt können wir auch eine Funktion, die auf Zeichenketten arbeitet, doppelt anwenden. Zum Beispiel, die Funktion, die das erste Auftreten der Ziffer `0` durch den Buchstaben `A` ersetzt.

```
twice((x)->x.replaceFirst("0","A"),"012304567890")
```

```
$6 ==> "A123A4567890"
```

Zweimal angewendet, werden so die ersten beiden Ziffern `0` ersetzt.

Aufgabe 1 In diesem Aufgabe soll nicht eine Funktion doppelt angewendet werden, sondern zwei übergebene Funktionen nacheinander. Also für zwei Funktionen f_1 und f_2 soll $f_2(f_1(x))$ gerechnet werden.

Jetzt sind drei Typen involviert. Ein beliebiger Argumenttyp `A` für die Funktion f_1, der Ergebnistyp `B` der Funktion f_1, der gleichzeitig Argumenttyp von f_2 ist und schließlich der Ergebnistyp `C` der Funktion f_1.

So ist die Funktion über drei Typen generisch zu halten: `<A, B, C>`.

Implementieren Sie die Funktion:

Funktionsdaten.java

```
static <A,B,C> C nacheinander(Function<A,B> f1,Function<B,C> f2, A a){
  return null; //TODO
}
```

Soll ein Argument erst quadriert und dann verdoppelt werden, dann kann folgender Ausdruck verwendet werden.

```
nacheinander((x)->x*x, (x)->2*x, 5)

$2 ==> 50
```

In diesem Fall sind alle drei Typen `A`, `B` und `C` vom konkreten Typ `Integer`.

Wir können aber auch Aufrufe machen, in denen alle drei Typen unterschiedlich sind:

```
nacheinander((x)->x.length(),(x)->Math.pow(2,x),"hallo")

$8 ==> 32.0
```

Hier ist der Typ `A` eine Zeichenkette vom Typ `String`. Von dieser wird die Länge berechnet. Wir erhalten für `B` den Typ `Integer`. Die Funktion `pow` schließlich berechnet eine Zahl vom Typ `Double`, womit der Typ `C` in diesem Beispiel ein `Double` ist.

4 Funktionen als Ergebnistyp

Genauso, wie eine Funktion andere Funktionen als Argument erhalten kann, kann eine Funktion auch eine Funktion als Ergebnis berechnen.

So lässt sich der aus der Mathematik bekannte Operator $\circ$ zur Verknüpfung zweier Funktionen in Java definieren.

$$(f_2 \circ f_1)(x) = f_2(f_1(x))$$

Der Operator $\circ$ verknüpft zwei Funktionen zu einer neuen. Er wird gerne als nach gelesen. $f_2 \circ f_1$ heißt also, wende f_2 nach f_1 an.

Wir erhalten folgende Umsetzung in Java:

Funktionsdaten.java

```
static  <A,B,C> Function<A,C> nach(Function<B,C> f2,Function<A,B> f1){
    return (x)->f2.apply(f1.apply(x));
}
```

Jetzt können wir eine neue Funktion erzeugen, indem wir zwei Funktionen zu einer konkatenieren:

```
Function<Integer,Integer> f = nach((x)->x*x, (x)->2*x)
```

```
f ==> $Lambda$24/0x0000000800c0aad0@506e1b77
```

Und getrennt davon die neu berechnete Funktion anwenden.

```
f.apply(5)
```

```
$5 ==> 100
```

Aufgabe 2 Betrachten Sie folgende Funktionsdefinition der Funktion, die nicht wie `twice` eine Funktion zweimal anwendet, sondern n mal.

$$\text{nTimes}(f, n, x) = \begin{cases} x & \text{für } n \leq 0 \\ \text{nTimes}(f, n-1, f(x)) & \text{für } n > 0 \end{cases}$$

Implementieren Sie die Funktion als generische Funktion in Java.

Funktionsdaten.java

```
static <A> A nTimes(Function<A,A> f,int n,A x){
  return null; //TODO
}
```

5 Weitere Funktionsargumente

5.1 Binäre Funktionen

Die Funktionenobjekte, die wir bisher gesehen haben, hatten alle genau ein Argument. Für zweistellige Funktionen verwendet Java den Typnamen `BiFunction`.[1]

Hier müssen drei Typen angegeben werden: Die beiden Argumenttypen und als drittes der Ergebnistyp.

Ein kleines Beispiel für eine zweistellige Funktion:

```
BiFunction<String,Integer,Integer> f
  = (String x,Integer y) -> x.length()+y
```

Sie addiert zur Länge einer Zeichenkette eine Zahl:

```
f.apply("hallo",3)
```

```
$8 ==> 8
```

Aufgabe 3 Schreiben Sie eine Funktion `zipWith`, die die Elemente zweier Listen paarweise mit einer binären Funktion verrechnet und in einer Ergebnisliste einfügt.

Funktionsdaten.java

```
static <A,B,C> AL<C> zipWith(BiFunction<A,B,C> f, AL<A> as, AL<B> bs){
  var cs = new AL<C>();
  for (var i=0;i<Math.min(as.size(),bs.size());i++){
     //TODO
  }
  return cs;
}
```

5.2 Binäre Operatoren

Ein Spezialfall binärer Funktionen sind solche, in denen beide Argumente vom gleichen Typ sind wie der Ergebnistyp. Eine solche binäre Funktion wird auch als Operator bezeichnet. Für diese kennt Java einen eigenen Typ: `BinaryOperator`.

Für eine Liste von Zahlen lässt sich mit Hilfe eines binären Operators eine sehr nützliche Funktion schreiben. Alle Elemente der Liste sollen mit einem Operator verknüft werden.

[1] Es gibt in Java bisher noch keinen allgemeinen Typ für beliebig stellige Funktionen. Die Typen von Funktionsobjekten und ihre Namen legen die Anwenderschnittstellen fest.

Funktionsdaten.java

```
static int verknuepfe(AL<Integer> xs, int r, BinaryOperator<Integer> op){
   for (var i=0;i<xs.size();i++) r=op.apply(r,xs.get(i));
   return r;
}
```

Die Summe aller Elemente der Liste lässt sich mit folgendem Ausdruck berechnen:

```
verknuepfe(xs,0,(x,y) -> x+y)

$4 ==> 55
```

Das Produkt aller Elemente der Liste lässt sich mit folgendem Ausdruck berechnen:

```
verknuepfe(is,1,(x,y) -> x*y)

$5 ==> 3628800
```

Und folgender Aufruf berechnet das größte Element einer Liste.

```
verknuepfe(is,Integer.MIN_VALUE,(x,y) -> Math.max(x,y))

$6 ==> 10
```

Die Funktion `verknuepfe` begegnet einem in der einen oder anderen Form in sehr vielen Programmiersprachen. Sie heißt dann oft `fold` oder `reduce`. In Java gibt es mehrere Funktionen `reduce`, die nach diesem Prinzip arbeiten. Wir werden sie später kennenlernen.

5.3 Prozedurobjekte

Auch für einstellige Prozeduren, also Funktionen mit einem Argument und dem Ergebnis `void` kennt Java einen Standardtyp. Er wird als `Consumer` bezeichnet. Sie konsumieren quasi ein Objekt, ohne ein Ergeb nis zu berechnen. Man kann solche Funktionsobjekte zum Beispiel nutzen, um für jedes Element einer Reihung eine bestimmte Aktion durchzuführen.

Damit lässt sich ausdrücken, dass für alle Elemente etwas zu tun ist:

Funktionsdaten.java

```
static <A> void fuerAlle(AL<A> xs, Consumer<A> c){
  for (var i=0;i<xs.size();i++) c.accept(xs.get(i));
}
```

Betrachten beispielsweise wir folgende Reihung von Zeichenketten:

```
var xs = of("Freunde","Römer","Landsleute")
```

Wir können jedes Element in Großbuchstaben auf der Kommandodzeile ausgeben, durch den Aufruf:

```
fuerAlle(xs,(x) -> System.out.println(x.toUpperCase()))
```

```
FREUNDE
RÖMER
LANDSLEUTE
```

5.4 Prädikate

Prädikate kennt man aus der Prädikatenlogik. Sie beschreiben dort eine Eigenschaft für Elemente auf einer Grundmenge. Ein Prädikat ist dann die Teilmenge, für die eine Eigenschaft gilt. Die Eigenschaft selbst kann man als eine Testfunktion verstehen. Die Testfunktion prüft, ob die Eigenschaft für ein bestimmtes Element gilt und wertet zu einen entsprechenden Wahrheitswert aus. Insofern kann ein Prädikat über diese Testfunktion charakterisiert werden. Somit ließe sich ein Prädikat auf der Menge A als Funktion des Typs: `Function<A,Boolean>` darstellen.

Java sieht hierfür eine spezialisierte Funktion vor mit dem Namen
`Predicate`.

Diese ist generisch über den Elementtyp, für den das Prädikat getestet witrd. Die eugentlich anzuwendende Funktion heißt hier nicht `apply` sondern passender Weise `test`.

Beispiel 5.3 Als Beispiel können wir quasi den Existenzquantor der Prädikatenlogik auf Listen als Funktion umsetzen. In der Prädikatenlogik ist eine Formel $\exists x.P(x)$ wahr, wenn mindestens ein Element der Grundmenge die Eigenschaft P hat.

Als Grundmenge nehmen wir eine Liste von Elementen und wollen wissen, ob ein Element der Liste das Prädikat erfüllt.

Funktionsdaten.java

```
static <A> boolean exists(AL<A> xs, Predicate<A> p){
  for (var i=0; i<xs.size();i++) if (p.test(xs.get(i))) return true;
  return false;
}
```

Existiert in den Zahlen von 1 bis 10 eine durch 5 teilbare Zahl?

```
exists(xs, (x) -> x%5==0)
```

```
$613 ==> true
```

Existiert in den Zahlen von 1 bis 10 eine Zahl über 1000?

```
exists(xs, (x) -> x>1000)
```

```
$612 ==> false
```

Naheliegender Weise will man vielleicht auch eine Entsprechung des Allquantors für die Elemente einer Liste implementieren.

Funktionsdaten.java

```
static <A> boolean all1(AL<A> xs, Predicate<A> p){
  for (var i=0; i<xs.size();i++) if (!p.test(xs.get(i))) return false;
  return true;
}
```

Mann kann sich aber auch einer logischen Äquivalenz bedienen, die den Allquantor über den Existenzquantor ausdrücken kann:

$$\forall x.P(x) \equiv \neg\exists x.\neg P(x)$$

Diese Äquvalenz lässt sich umgangssprachlich ausdrücken: Wenn es kein Element gibt, das die Eigenschaft verletzt, dann müssen wohl alle die Eigenschaft haben.

Da auf den Funktionsobjekten des Typs `Predicate` eine Methode zum Negieren eines Prädikats existiert, kann diese Äquivalenz auch direkt in Java ausgedrückt werden.

Funktionsdaten.java

```
static <A> boolean all(AL<A> xs, Predicate<A> p){
  return !exists(xs, p.negate());
}
```

Aufgabe 4 Schreiben Sie eine generische Funktion `filter`, die für eine Liste eine Ergebnisliste von all den Elementen der Liste erzeugt, für die ein übergebenes Prädikat gilt.

Funktionsdaten.java

```
static <A> AL<A> filter(AL<A> xs, Predicate<A> p){
  var rs = new AL<A>();
  //TODO
  return rs;
}
```

Beispielaufruf, der nach allen durch 3 teilbahren Zahlen filtert:

```
filter(xs,(x)->x%3==0)
```

```
$617 ==> [3, 6, 9]
```

5.5 Vergleiche

Eine häufig benötigte Funktion ist der Vergleich von zwei Objekten gleichen Typs in Bezug auf eine Ordnungsrelation: die Frage also, welches von zwei Objekten als größer bzw. kleiner angesehen wird oder ob die beiden Objekte als gleich betrachtet werden. Eine Ordnungsrelation spielt eine entscheidene Rolle bei unserer Umsetzung von Mengen als binären Suchbaum. Bei einem binären Suchbaum sind im linken Teilbaum alle gespeicherten Elemente klainer als das Element an der Wurzel und im rechtem Teilbaum alle Element gößer als das Wurzelelement. Wenn man eine generische Implementierung über die Elemente der Menge hat, benötigt man eine Vergleichsoperation auf den Elementen des variablen Typs.

Funktionsdaten.java

```
record BT<E>(BT<E> left, E e, BT<E> right){}
```

Zunächst die Funktion, die ein Blatt erzeugt, also eine einelementige Menge:

Funktionsdaten.java

```
static <E> BT<E> blatt(E e){return new BT<>(null,e,null);}
```

Die Größe einer Menge bleibt vom Typen der Elemente unberührt:

Funktionsdaten.java

```
static <E> long size(BT<E> t){
  return t == null ? 0 : 1+size(t.left())+size(t.right());
}
```

Interessant wird die Funktion, die ein neues Element in die Menge einfügen soll. Hier gilt die Regel: ist das einzufügende Element kleiner als das Wurzelelement, ist es im linken Teilbaum einzufügen, ist es größer, dann im rechten Teilbaum. Aber was heißt für einen beliebigen aber festen Typen `E`, der variabel gehalten ist, größer bzw. kleiner?

Das können wir nicht erraten und diese Entscheidung muss eine zusätzliche Funktion treffen, die zwei Objekte vergleichen kann.

Die Funktion zum Vergleich zweier Objekte in Java heißt `Comperator`. Sie ist generisch über den Typ der Elemente, die verglichen werden sollen. Der Funktionsname lautet `compare`. Sie hat zwei Argumente des Elementtyps, der zu vergleichenden Elemente. Das Ergebnis ist vom Typ `int`. Stellt das Ergebnis eine negative Zahl dar, so ist das erste Element als kleiner anzusehen, bei einr positiven Zahl als größer und beim Ergebnis 0 sind beide Elemente als gleich groß zu betrachten.

Um das mit Leben zu füllen, definieren wir eine kleine Klasse für Punkte im zweidimensionalen Raum.

Funktionsdaten.java

```
record Punkt(double x, double y){}
```

Für Objekte dieser Klasse können wir eine Vergleichsfunktion definieren. Dieses wird eine Funktion des Typs `Comparator<Punkt>`. Diese lässt sich durch einen Lambda-Ausdruck definieren:

Funktionsdaten.java

```
static Comparator<Punkt> pc = (p1,p2) ->
  (p1.x()<p2.x() ? -1 :
   p1.x()>p2.x() ? 1  :
   p1.y()<p2.y() ? -1 :
   p1.y()>p2.y() ? 1  :
   0);
```

Beim Vergleich zweier Punkte sortieren wir mit dieser Funktion erst nach der x-Koordinate, bei gleicher x-Koordinate nach der y-Koordinate und erst wenn beide gleich sind, sehen wir die beiden Punkte auch als gleich an.

Nun zurück zu unseren binären Suchbaum. Die Funktion `add` benötigt einen Vergleich auf die Elemente der Menge. Deshalb bekommt die die Funktion zum Vergleichen zweier Elemente als zusätzliches Argument.

Funktionsdaten.java

```
static <E> BT<E> add(BT<E> t, E e, Comparator<E> comp){
  return
    t == null ? blatt(e):
    comp.compare(e,t.e())<0
              ? new BT<>(add(t.left(),e,comp),t.e(),t.right()):
    comp.compare(e,t.e())>0
              ? new BT<>(t.left(),t.e(),add(t.right(),e,comp)):
    t;
}
```

Wenn es sich bei den Elementen der Menge um Objekte der Klasse `Punkt` handelt, dann überladen wir die Funktion `add` am besten so, dass die obige Vergleichsfunktion `pc` verwendet wird.

Funktionsdaten.java

```
static BT<Punkt> add(BT<Punkt> t, Punkt e){return add(t,e,pc);}
```

Ebenso verfahren wir bei der Funktion zum Testen, ob ein Element in der Menge enthalten ist. Auch diese benötigt die Vergleichsfunktion für die Elemente als zusätzliches Argument.

Funktionsdaten.java

```
static <E> boolean contains(BT<E> t, E e, Comparator<E> comp){
  return
    t == null ? false:
    comp.compare(e,t.e())<0 ? contains(t.left(),e,comp):
    comp.compare(e,t.e())>0 ? contains(t.right(),e,comp):
                true;
}
```

Um auch hier für den Vergleich zweier Objekte der Klasse `Punkt` die stets gleiche Funktion zum Vergleichen zu verwenden, kann man auch diese Funktion für die Menge von Punktobjekten überladen.

```
boolean contains(BT<Punkt> t, Punkt e){return contains(t,e,pc);}
```

Aufgabe 5 Schreiben Sie eine Funktion, die die Elemente des binären Suchbaums in aufsteigender Form in eine Liste einträgt.

Funktionsdaten.java

```
static <A> AL<A> toSortedList(BT<A> t){
  return toSortedList(t,new AL<A>());
}

static <A> AL<A> toSortedList(BT<A> t, AL<A> rs){
  //TODO
  return rs;
}
```

6 Eigene Funktionstypen

Bisher haben wir vordefinierte Funktionstypen verwendet. Wir können aber auch eigene Funktionstypen definieren, die einen eigenen Namen haben und auch mit einen eigenen Namen für die eigentliche Anwendung der Funktion ausgestattet sind.

Der Weg dahin geht über die Schnittstellen. Funktionstypen sind in Java Schnittstellen mit genau einer abstrakten Methode.

Ein Typ für binäre Operatoren auf Wahrheitswerten lässt sich definieren als:

```
interface BooleanOp{
  boolean eval(boolean b1, boolean b2);
}
```

Wenn eine Schnittstelle nur eine abstrakte Methode definiert, dann können wir mit Lambda definierte Funktionen für diese Schnittstelle schreiben.

```
BooleanOp implication = (b1,b2) -> !b1||b2

implication ==> $Lambda$22/0x0000000800c0e610@484b61fc
```

Diese kann nun mit dem Methodennamen `eval` aufgerufen werden.

```
implication.eval(false,true)
```

```
$31 ==> true
```

```
implication.eval(true,false)
```

```
$32 ==> false
```

Die Lambda-Ausdrücke sind nur eine sehr abkürzende Schreibweise dafür, dass eine Datenklasse erzeugt wird, die die Schnittstelle implementiert.

```
record Implication() implements BooleanOp{
  public boolean eval(boolean b1, boolean b2){
    return !b1||b2;
  }
}
```

Anschließend kann von dieser Datenklasse ein Objekt erzeugt werden:

```
BooleanOp implication = new Implication()
```

```
implication ==> Implication[]
```

Wie man sieht ist die Lambda-Notation wesentlich kompakter. Es braucht nicht die zwei getrennte Schritte, erst eine Datenklasse zu definieren und dann von dieser ein Objekt zu erzeugen. Der Lambda-Ausdruck macht beides in einem. So braucht auch kein Namen für die Datenklasse, von der ja nur ein Objekt erzeugt werden soll, ausgedacht werden. Man muss beim Schreiben des Lambda-Ausdrucks noch nicht einmal wissen, wie die abstrakte Methode der Schnittstelle heißt. Statt eines Funktionsnamens schreibt man nur den stilisierten Pfeil `->`.

6.1 Annotation für Funktionsschnittstellen

Man kann Funktionsschnittstellen noch zusätzlich mit einer sogenannten Annotation als Funktion markieren. Annotationen beginnen in Java mit dem Klammeraffensymbol `@`. Sie können Programmteile mit zusätzlichen Informationen versehen. Die Annotation für Funktionsschnittstellen lautet: `@FunctionalInterface`.

Mit dieser zusätzlichen Annotation erhält man keine zusätzliche Funktionalität im Sinne der Auswertung des Programms. Man erhält aber eine vorherige statische Prüfung, die noch einmal checkt, ob die Schnittstelle tatsächlich nur eine abstrakte Methode hat und somit genau eine Funktion repräsentiert, deren Namen und Signatur sich dann automatisch aus dem Kontext ergeben können. Wird diese Eigenschaft verletzt, weist Java das Programm im Vorfeld mit einer entsprechenden Fehlermeldung zurück.

```
@FunctionalInterface interface A{
  int apply1(int i);
  String apply2(int i);
}
```

```
|  Error:
|  Unexpected @FunctionalInterface annotation
|    A is not a functional interface
|      multiple non-overriding abstract methods found in interface A
|  @FunctionalInterface interface A{
|  ^------------------^
```

6.2 Generische Funktionsschnittstellen

Es ist natürlich auch möglich eine Funktionsschnittstelle generisch zu gestalten.

Beispiel 6.4 Hier ist zum Beispiel eine beliebige dreistellige Funktion generisch umgesetzt.

```
@FunctionalInterface interface Fun3<A,B,C,R>{
  R apply(A a,B b,C c);
}
```

Diese lässt sich jetzt für unterschiedliche Typen der Typvariablen mit Lambda ausdrücken instanzieren.

Vielleicht mit drei ganzen Zahlen und einem ganzzahligen Ergebnis.

```
Fun3<Integer,Integer,Integer,Integer> f1 = (x,y,z)->x+y*z
```

Das entsprechend anzuwenden ist.

```
f1.apply(21,3,7)
```

```
$15 ==> 42
```

Oder aber mit einem Argument als Zeichenkette und einer Zeichenkette als Ergebnis.

```
Fun3<String,Integer,Integer,String> f2 = (x,y,z)->x+y+"*"+z+"="+y*z
```

Damit lassen sich dann entsprechdne Rechnungen durchführen.

```
f2.apply("Die Rechnung lautet:",3,17)
```

```
$17 ==> "Die Rechnung lautet:3*17=51"
```

7 Anwendungen von Funktionsdaten

In diesem Abschnitt zeigen wir an ein paar kleinen Beispielen, wie mächtig das Prinzip der Funktionsdaten ist.

7.1 Evaluationszeit Messen

Beginnen wir damit, dass wir an der Ausführungszeit eine Auswertung interessiert sind.

Hierzu haben wir das Paket importiert, das Funktionen und Objekten zur Zeitmessung bereitstellt.

Mit dem Ausdruck `Instant.now()` lässt sich ein aktueller Zeitstempel ermitteln.

```
var z1 = Instant.now()

z1 ==> 2022-12-25T11:32:03.208354646Z
```

Der gleiche Aufruf eine knappe Minute später gibt natürlich einen späteren Zeitpunkt.

```
var z2 = Instant.now()

z2 ==> 2022-12-25T11:32:57.348488191Z
```

Mit der Funktion `Duration.between` lässt sich die Zeitspanne zwischen zwei Zeitpunkten errechnen:

```
var d = Duration.between(z1,z2)

d ==> PT54.140133545S
```

Jetzt haben wir soweit alles, was wir brauchen, um zu messen, wie lange die Auswertung eines Ausdrucks benötigt.

Jetzt können wir versuchen eine Funktion zu schreiben, die einen Ausdruck erhält, der auszuwerten ist und ein Paar errechnet, das aus dem Ergebnis der Auswertung und der Auswertungszeitdauer besteht. Hierzu sei wieder die generische Paarklasse verwendet.

Funktionsdaten.java

```
record Pair<A,B>(A e1,B e2){}
```

Angenommen, wir haben einen Ausdruck `e` von einem generisch gehaltenen Typen `A`. Dann wollen wir `e` auswerten, und die Zeitdauer von vor und nach der Auswertung messen. Das soll in einer Funktion `timeMeasure` geschehen mit dem Rückgabetyp: `Pair<A,Duration>`.

Die Funktion kann nicht den Ausdruck `e` als Parameter bekommen, denn wäre der Ausdruck ja bei Übergabe an die Funktion ausgewertet und würde nicht erst in der Funktion ausgewertet. Wir können aber eine funktionale Schnittstelle vorsehen, die

Funktionsdaten.java

```
@FunctionalInterface interface Getter<A> {
  A get();
}
```

Diese Schnittstelle stellt eine konstante Funktion dar für die Auswertung eines Ausdrucks eines variabel gehaltenen Typs `A`. Ein Objekt des Typs `Getter<A>` lässt sich durch einen Lambda-Ausdruck leichht erstellen:

```
Getter<Long> gl = () -> fib(5)
```

Damit haben wir die Möglichkeit, nicht den Wert eines Ausdrucks zu übergeben, sondern die Funktion, die diesen Ausdruck erst berechnet, wenn die Methode `get` aufgerufen wird.

So lässt sich eine Funktion, die einen beliebigen Ausdruck auswertet und zusätzlich die Auswertungsdauer misst definieren. Es wird ein Zeittempel erzeugt, die Berechenung des Ausdrucks angestoßen und anschließend ein zweiter Zeitstempel erzeugt. Zeitdauer und Auwertungsergebnis werden in einem Paarobjekt zusammengefasst.

Funktionsdaten.java

```
static <A> Pair<A,Duration> timeMeasure(Getter<A> c){
  var start = Instant.now();
  A r = c.get();
  var end = Instant.now();
  return new Pair<>(r, Duration.between(start, end));
}
```

Betrachten wir als Beispiel wieder die Fibonaccifunktion:

Funktionsdaten.java

```
static long fib(int n){
  return n<=1 ? n : fib(n-2)+fib(n-1);
}
```

Nun können wir nicht nur die Funktion aufrufen, sondern zusätzlich die Auswertungszeit messen.

```
timeMeasure( () -> fib(50) )

$40 ==> Pair[e1=12586269025, e2=PT46.255992979S]
```

7.2 Eine Funktion zur parallelen Auswertung

Wir haben bereits das Teile-und-Herrsche-Verfahren als algorithmisches Prinzip betrachtet. Ein Beispiel war dabei die Funktion, die das Produkt der zahlen aus einem Zahlenbereich errechnet.

Funktionsdaten.java

```
static long prod(int from, int to){
  var middle = (from+to)/2;
  return from==to ? from
                  : prod(from,middle) * prod(middle+1,to);
}
```

Das Teile-und-Herrsche-Prinzip kann seine Stärke nur ausspielen, wenn die beiden Teilaufgaben, hier also die Berechnung von `prod(from,middle)`und `prod(middle1,to)`+ auch unabhängig und parallel voneinander stattfinden kann. Auch hierzu müssen die beiden parallel auszuwertenden Ausdrücke in unausgewerteter Form übergeben werden und auch hier kann man sich eines konstanten Funktionsobjektes bedienen.

In Java gibt es das Paket `java.util.concurrent` mit Funktionalität zur parallelen Auswertung. Unserer Schnittstelle `Getter` entsprechend gibt es in diesem API eine funktionale Schnittstelle `Callable`.

Mit Hilfe der dort angebotenen Klassen, lässt sich eine Funktion definieren, die zwei unasugewertete Ausdrücke erhält, die je ein Teilergebnis berechnen, so wie eine zweistellige Funktion, die diese Teilergebnisse zu einem Gesamtergebnis zusammenführt. Die beiden Teilaufgaben werden dann parallel ausgeführt und anschließend zusammengeführt.

So lässt sich eine Funktion `par` definieren:

Funktionsdaten.java

```
static <A,B,C> C par(Callable<A> pA, Callable<B> pB, BiFunction<A,B,C> comb){
  try (var executor  = Executors.newFixedThreadPool(2)){
    var p1 = executor.submit(pA);
    var p2 = executor.submit(pB);
    var r  = comb.apply(p1.get(),p2.get());
    executor.shutdown();
    return  r;
  } catch (Exception e) {
    throw new RuntimeException(e);
  }
}
```

Mit dieser Funktion, lässt sich jetzt die Berechnung des Produktes parallelisiere als Funktion `prodP`.

Funktionsdaten.java

```
static long prodP(int from, int to){
  var middle = (from+to)/2;
  return from==to
    ? from
    : par( () -> prod(from,middle)
         , () -> prod(middle+1,to)
         , (x,y) -> x*y);
}
```

Wir übergeben der Funktion `par` die beiden rekursiven Aufrufe als unausgewertete Ausdrücke des Typs `Callable`. Zusätzlich die Multiplikationsfunktion, die die beiden Teilergebnisse verrechnen soll. Im rekursiven Aufruf verwenden wir nicht die parallele Version, sondern die ursprüngliche sequentielle Version.

Nun stellt sich die Frage, ob wir durch die Parallelisierung der beiden rekursiven Aufdrufe tatsächlich die Ausführungszeit verkürzen können. Potentiell ist fast eine Halbierug der Auswertungszeit möglich.

Zunächst ein Aufruf der nicht parallelisierten Version:

```
timeMeasure( ()->prod(1,25) )

Pair[e1=7034535277573963776, e2=PT0.000030443S]
```

Und nun die parallelisierte Version:

```
timeMeasure( ()->prodP(1,25) )

Pair[e1=7034535277573963776, e2=PT0.002682844S]
```

Die Parallelisierung macht es nicht schneller, sondern ein Vielfaches langsamer. Das liegt daran, dass das Verteilen der Arbeit auf parallele Ausführung nicht umsonst ist. Eine Parallelisierung lohnt erst dann, wenn die parallel ausgeführten Teile aufwändiger sind, als der Zusatzaufwand der Parallelisierung.

Betrachten wir jetzt eine teilweise parallelisierte Version der Baunrekursion zur Berechnung einer Fibonaccizahl. Hier sehen wir einen zusätzlichen Parameter, der angibt bis zu welcher Baumtiefe noch parallelisiert werden soll, bevor auf die komplett sequentielle Variante zurückgegriffen wird.

Funktionsdaten.java

```
static long fibP(int n,int d){
  return d<=0 || n<=1
          ? fib(n)
          : par( () -> fibP(n-2,d-1)
               , () -> fibP(n-1,d-1)
               , (x,y)->x+y);
}
```

Jetzt lässt sich eine deutliche Verkürzung der Auswertungszeit auch messen. Zunächst die parallele Version:

```
timeMeasure(() -> fibP(45,3))

Pair[e1=1134903170, e2=PT2.202314362S]
```

Im Vergleich dazu die nicht parallelisierte Version.

```
timeMeasure(() -> fib(45))

Pair[e1=1134903170, e2=PT6.72789178S]
```

Es wird in diesem Fall nur noch ein Drittel der sequentiellen Auswertungszeit benötigt.

Aufgabe 6 Betrachten Sie die Funktion, die für einen Binärbaum testet, ob ein Element mit einer bestimmten Eigenschaft enthalten ist.

Funktionsdaten.java

```
static <A> boolean containsWith(BT<A> t,Predicate<A> p){
  if(t==null) return false;
  if (p.test(t.e())) return true;
  if (containsWith(t.left(),p)) return true;
  return containsWith(t.right(),p);
}
```

Schreiben Sie eine parallele Variante dieser Funktion:

Funktionsdaten.java

```
static <A> boolean containsWithPar(BT<A> t,Predicate<A> p){
  if(t==null) return false;
  if (p.test(t.e())) return true;
  return false; //TODO
}
```

Überlegen Sie, ob und wenn wann die Parallelisierung sinnnvoll ist und wann sie kritisch zu bewerten ist.

7.3 Nicht-strikte Funktion mit Funktionsargumenten

Wir haben bereits festgestellt, dass Java eine strikte Auswertungsstrategie verfolgt. Jedes Argument wird erst komplett ausgewertet, bevor es der Funktion übergeben wird.

Wir können mit Funktionsargumenten eine nicht-strikte Auswertung simulieren. Statt den eigentlichen Argumenten übergeben wir eine nullstellige Funktion, die das Argument nach Bedarf auswerten kann.

Hierzu können wir uns wieder eine generische Funktionsschnittstelle definieren, in der eine Funktion zum Auswerten des Ergebnisses vorgesehen ist.

Funktionsdaten.java

```
interface Lazy<A> {
  A eval();
}
```

Unser Beispiel für die Striktheit von Funktionen war die Funktion `wenn`, die die Fallunterscheidung macht. Diese braucht nur nach Bedarf eines der beiden Argumente für die Alternativen auswerten. Statt jetzt Argumente des generischen Typs für diese vorzusehen, sehen wir Funktionen des Typs `Lazy` für diese vor.

Wir übergeben also Funktionen, die durch Aufruf der Methode `eval` die Argumente erst auswerten.

Diese können nun dem Bedingungsoperator übergeben werden. Erst dort wird die Auswertung nach Bedarf mit dem Aufruf von `eval()` angestoßen.

Funktionsdaten.java

```
static <A> A wenn(boolean b, Lazy<A> a1, Lazy<A> a2) {
  return b ? a1.eval() : a2.eval();
}
```

Wir können uns wieder der nichtterminierenden Fehlerfunktion `bot()` bedienen, um zu zeigen, dass nun die Argumente erst nach Bedarf ausgewertet werden. Hierzu sind statt der Argumente direkt Lambda-Ausdrücke zu übergeben. Ein Lambda-Ausdruck für eine Funktion des Typs `Lazy` beginnt stets mit: `()->`. Es handelt sich ja um eine Funktion ohne Argumente.

Der Aufruf zu `wenn` wird also:

```
wenn(2+17==24, ()->bot(), ()->"Nein")
```

```
$629 ==> "Nein"
```

Und wie man sieht, terminiert seine Auswertung, obwohl im zweiten Argument ein Aufruf der nichtterminierenden Funktion `bot()` steht.

Wir können die Funktion `wenn` auch so überladen, dass sie strikt und nicht-strikt aufgerufen werden kann.

Funktionsdaten.java

```
static <A> A wenn(boolean b, A a1, A a2) {
  return wenn(b, ()->a1, ()->a2);
}
```

Aufgabe 7 Schreiben Sie eine Funktion, die die logische Implikation realisiert und so gut es geht, vermeidet, die Argumente auszuwerten.

Funktionsdaten.java

```
static boolean implication(Lazy<Boolean> a1, Lazy<Boolean> a2) {
  return false; //TODO
}
```

In der Java Standardbibliothek gibt es eine Funktionsschnittstelle, die unserer Schnittstelle `Lazy` entspricht. Dort heißt sie `Supplier`. Sie ist ebenso generisch: Dort heißt die Methode nicht `eval` wie bei uns, sondern `get`.

Zum Abschluss dieses Lehrbriefes ist noch der Rumpf der ganz am Anfang definierten Schnittstelle `Funktionsdaten` zu schließen.

Funktionsdaten.java

```
}
```

Literatur

[Chu36] Alonzo Church. An Unsolvable Problem of Elementary Number Theory. American Journal of Mathematics, 58(2):345–363, April 1936.

[Tur36] A. Turing. On Computable Numbers, With an Application to the Entscheidungsproblem. In Proceedings of the London Mathematical Society, volume 42 of 2, 1936.

[Tur37] A. Turing. Computability and λ-Definability. The Journal of Symbolic Logic, 4, 1937.