Funktionale Programmierung mit JS

Donnerstag, 17. Januar 2019 09:51

Einführung in die Funktionale Programmierung mit JavaScript

Contents

Chapter 1

Historie

- 1930er: λ-Kalkül (Alonzo Church): formale Sprache, zur Beschreibung von Funktionen und deren gebundeme Parameter
- Church-Turing-These: λ-Kalkül und Turing-Maschine sind ebenbürtig im Sinne der Berechenbarkeit
- 1950er: Entwicklung erster funktionaler Programmiersprachen (Lisp)

Chapter 2

Motivation

Neue Laufzeitumgebungen (Virtualisierung, Cloud, Containerisierung, ...) stellen besondere Anforderungen an Anwendungen:

2.1 Modularisierung

- perfekte Wiederverwendbarkeit
- 100%-ige Testabdeckung

2.2 Parallelisierung

- komplette Unabhängigkeit vom Kontext (Zustand, Server, Laufzeitumgebung, ...)

2.3 Neue Art von Anwendungen

- Big Data, Data Science
- Streaming

Funktionale Programmierung bietet genau das! => hört sich nach einem Wundermittel an!

Chapter 3

Grundlegende Konzepte

3.1 (De-)Komposition

- (De-)Komposition ist keine Anforderung der Computer!
- Es gilt die 7 +/- 2 Regel
- Eleganter Code:= Komponenenten sind von einer Größe, die gerade noch vom menschl. Geist erfasst werden können
- Black-Box: "Vergiss die Implementierung"

3.2 Stringente Mathematische Theorie (Kategorientheorie)

- es ist schwierig/unmöglich zu beweisen, dass ein Programm korrekt funktioniert
- Programmierer verwenden beim Programmieren einen "Interpreter" im Kopf -> funktioniert eher schlecht!
- Abhilfe durch "Denotationelle Semantik": Programmkonstrukt -> mathem. Interpretation
- Mathematische Beweise können Menschen relativ gut! (Erfahrung über mehrere tausend Jahren)

Chapter 4

Prinzipien

4.1 Funktionen als "First-Class-Citizens"

- Speicherung der Funktionen in einer Variablen
- Funktionen als Übergabe-/Rückgabeparameter von Funktionen (Higher-order function, HOF)
- Erstellung von Funktionen zur Laufzeit
- Anonyme Funktionen

4.2 Referential Transparency (Referenzielle Transparenz)

- Ein Funktionsausdruck kann im Programm **immer** durch seinen Wert ersetzt werden
- Funktionen liefern bei gleichen Übergabeparametern **immer** das selbe Ergebnis

=> Funktionen sind *cacheable* (Memoisation)
Achtung: Referenzielle Transparenz $\neq$ Idempotenz!

4.3 Funktionen haben keine Side-Effects (Wirkung)

- verwendete Variablen, Objekte, Parameter sind immer zustandslos und unveränderbar
- vgl. Value Objects bei DDD
- Gegenbeispiel: Entities bei DDD
- Keine Interaktion der Funktionen mit dem Kontext (Gegenbeispiel: CCC in AOP)

4.4 Pure Functions

Funktionen die sowohl - referenziell transparent sind - keine Side-Effects haben,
nennt man *Pure Functions*.

Chapter 5

Techniken

5.1 In Objekt-Orientierten Sprachen

- Alle Variablen als `final` deklarieren (Achtung: referenzierte Objekte konnen mutable sein)
- Klassen als `final` deklarieren (verhindert Vererbung und Überschreiben von Methoden)
- Kein Default-Konstruktor, *ein* definierter Konstruktor
- keine `set`-Methoden

=> Objekte dienen als reine Datencontainer
=> Klonen der Objekte bei (beabsichtigter) Status-Änderung (vgl. *State-Pattern*)

5.2 Konstrukte Funktionaler Sprachen am Beispiel JavaScript

5.2.1 Warum JavaScript?

- Grundlagen sollten allen Teilnehmern bekannt sein
- wachsende Bedeutung der Programmiersprache
- JavaScript wurde von der Programmiersprache *Scheme* beeinflusst, welche wiederum ein Lisp-Dialekt ist (Douglas Crockford: "Lisp in C's Clothing")
- JavaScript unterstützt viele Prinzipien der FP (Beispiele folgen)

=> "JavaScript is the world's most misunderstood programming language" - D. Crockford

5.2.2 Grundlagen

Zustandslose Datentypen

JavaScript bietet folgende Möglichkeiten Daten zustandslos zu machen.

- Verwendung von `const` bei *primitiven Datentypen*

```
In [1]: {
            const val = 2;
            // val = 3
            // val++
            val;
        }
```

```
Out[1]: 2
```

- Achtung: `const` funktioniert nicht bei Objekten!

```
In [2]: {
            const obj = {'val1' : 12};
            // obj = {'val3' : 26};
            obj.val1 = 25
            obj.val2 = 27
            obj;
        }
```

```
Out[2]: { val1: 25, val2: 27 }
```

- `Object.freeze(<object>)` erlaubt ein *shallow freeze*

```
In [3]: {
            const obj = {'val1' : 12};
            Object.freeze(obj);
            obj.val2 = 17
            obj;
        }
```

```
Out[3]: { val1: 12 }
```

```
In [4]: {
            const obj = {'val1' : 12, 'innerObj' : {'val2' : 17}};
            Object.freeze(obj);
            obj.innerObj.val2 = 23
            obj.innerObj.val2;
        }
```

Friert nur das Objekt ein, aber nicht die Ebene tiefer

```
Out[4]: 23
```

- 'Einfrieren' von Arrays

```
In [5]: {
            const arr = [1,2,3];
            Object.freeze(arr);
            // arr.push(4);      // führt zu Fehlermeldung
            arr[1] = 5;          // wird nicht ausgführt
            arr;
        }
```

```
Out[5]: [ 1, 2, 3 ]
```

Zustandslosigkeit durch Klonen von Objekten und Arrays

- Erzeugen von Objekt-Kopien mit `Object.assign()` (shallow copy)

```
In [6]: {
            const obj = {'val1' : 12, 'innerObj' : {'val2' : 17}};
            const newObj = Object.assign({}, obj);
            newObj.innerObj.val2 = 23
            // newObj;
            obj;
        }
```

kopiert nur das Objekt, aber referenziert auf dieselben member.

```
Out[6]: { val1: 12, innerObj: { val2: 23 } }
```

- Erzeugen von Objekt-Kopien mit dem spread operator `(...)` (shallow copy)

```
In [7]: {
            const obj = {'val1' : 12, 'innerObj' : {'val2' : 17}};
            const newObj = {...obj};
            newObj.innerObj.val2 = 23
            // newObj;
            obj;
        }
```

```
Out[7]: { val1: 12, innerObj: { val2: 23 } }
```

- Erzeugen von Array-Kopien mit `slice()` (shallow copy)

```
In [8]: {
            const arr = [1,2,3];
            const newArr = arr.slice()
            arr.push(4);
            newArr;
        }
```

```
Out[8]: [ 1, 2, 3 ]
```

- Erzeugen von Array-Kopien mit dem spread operator `(...)` (shallow copy)

```
In [9]: {
            const arr = [1,2,3];
            const newArr = [...arr]
            arr.push(4);
            newArr;
        }
```

```
Out[9]: [ 1, 2, 3 ]
```

Syntax zur Deklaration von Funktionen

Funktionen können in JavaScript auf unterschiedliche Art und Weise deklariert werden.

- "Klassische" Deklaration

```
In [10]: // "classic" way
         function mult(a, b) {
            return a * b;
         }
         // call
         mult(2,3);
```

```
Out[10]: 6
```

- Funktionsliterale: Speichern einer anonymen Funktion (s.u.) in einer Variablen

```
In [11]: // function literal
         var mult_fl = function(a,b) {
            return a * b;
         }
         // call
         mult_fl(3,5);
```

```
Out[11]: 15
```

- Lambdas / Pfeilfunktionen bieten eine verkürzte Schreibweise für Funktionsliterale

```
In [12]: // arrow functions
         var mult_ar = (a, b) => a * b;

         // complex lambda using {}: should be avoided!!!
         var calc = (x, y) => {
            const p = x*x;
            const q = 4*x*y;
            const r = 2*x;
            return (p-q)/r;
         };

In [13]: // call mult_arr
         mult_ar(4,3);

Out[13]: 12

In [14]: // call calc
         calc(2,3);

Out[14]: -5
```

Funktionen höherer Ordnung (Higher-order functions, HOFs)

Funktionen höherer Ordnung können - Funktionen als Parameter übergeben werden
und/oder - Funktionen als Rückgabewert liefern.

- Beispiel: Übergabe einer Funktion als Parameter

Hierdurch kann hier eine generische Funktion `applyOp` definiert werden, der je nach Anwendung eine konkrete Funktion (im Parameter `op`) übergeben wird.

```
In [15]: // pass function as argument
         var mult = (a, b) => a * b;
         var add = (a, b) => a + b;

         var applyOp = (a, b, op) => op(a,b);

In [16]: applyOp (2, 3, mult);

Out[16]: 6

In [17]: applyOp (2, 3, add);

Out[17]: 5
```

- Beispiel: Funktion als Rückgabewert.

Der Aufruf von `divideBy(3)` erzeugt zur Laufzeit eine (anonyme) Funktion in deren Umfeld `b=3` gesetzt ist. Durch die zweite Klammer (`(15)`) wird diese dann ausgeführt.

```
In [18]: // return functions from other functions
         // creation at runtime
         var divideBy = function(a) {
            return function (b) {
               return b / a;
            };
         }
         // generate new function
         var divide_by_3 = divideBy(3);
         // call function
         divide_by_3(15);
```

```
Out[18]: 5
```

```
In [19]: // even simpler call
         divideBy(3)(15);
```

```
Out[19]: 5
```

Anonyme Funkionen und direkte Ausführung

Funktionen müssen in JavaScript nicht mit Namen versehen werden. Häufig werden die so genannten anonymen Funktionen als Callback-Funktionen (asynchrone Ausführung!) oder als direkt ausfeführte anonyme Funktionen (immediately-invoked anonymous function expression (IIFE)) verwendet.

- Beispiel: Anonyme Funktion als Callback-Funktion

```
// anonymous functions
// used as calback
setInterval(function() {
   console.log('hello');
}, 10000);
```

- Beispiel: Anonyme Funktion als Funktionsliteral

Durch die Zuweisung wird die Funktion in einer Variablen gespeichert. Dadurch ist `add` vom Typ `function`.

```
In [20]: // anonymous function as function literal
         var add = function(x,y) {return x + y;};
```

```
In [21]: typeof(add)
```

```
Out[21]: 'function'
```

```
In [22]: add(1, 2)
```

```
Out[22]: 3
```

- Beispiel: Direkte Ausführung von anonymen Funktionen

Durch Anfügen der Übergabeparameter in `()`-Klammern an das Funktionsliteral wird die Funktion direkt ausgeführt. Dadurch ist `sum` vom Typ `number`.

```
In [23]: //anonymous function with direct execution
         var sum = (function(x, y) {return x + y;})(3, 4);
```

```
In [24]: typeof(sum)
```

```
Out[24]: 'number'
```

```
In [25]: sum
```

```
Out[25]: 7
```

- Beispiel: Direkte Ausführung von anonymen Funktionen ohne Zuweisung zu Variablen

Die Ausführung einer Funktion ist unabhängig von der Zuweisung zu einer Variablen. Sobald hinter dem Funktionsliteral die Übergabeparameter in ()-Klammern stehen, wird die Funktion ausgeführt. Man redet dann auch von **Immediately-Invoked Anonymous Function Expressions (IIFE)**.

```
In [26]: // immediately-invoked anonymous function expression (IIFE)
         (function(n) {
            return n*n ;
         })(6);                                  //  (function () { ... })()
```

```
Out[26]: 36
```

- Beispiel: IIFE - alternative Syntax

```
In [27]: //immediately-invoked anonymous function expression (IIFE)
         (function(n) {
            return n*n ;
         }(7));           // Douglas Crockford's style (function () { ... }())
```

```
Out[27]: 49
```

Prädikate

Prädikate sind (meist einfache) Funktionen, die `true` oder `false` als Rückgabeparameter haben. Diese werden häufig als Filter eingesetzt. Namen der Prädikate beginnen meist mit `is` oder `has`.

- Beispiele für Prädikate

```
In [28]: //typical definition
         function isBigger10(n){
            if(n>10){
               return true;
            }
            return false;
         }

         function isOdd(n){
            if(n%2 !== 0){
               return true;
            }
            return false;
         }

         //-----------------------
         //simpler
         function isBigger5(n){
            return n > 5;
```

```
}

//-----------------------
//even simpler with lambda
var isEven = n => n%2 === 0;
```

- Prädikate können mit Hilfe von HOFs auch kombiniert werden

```
In [29]: //predicate combinators
         //p, p1, p2 are predicates
         function and(p1,p2) {
            return function(x,y) {
               return p1(x) && p2(y);
            };
         }

         //using lamdda expression
         var not = p => {
            return function(x) {
               return !p(x);
            };
         };

         //predicate combinators with lamdas: one-liner
         var and = (p1,p2) => {return (x,y) => p1(x) && p2(y);}
         var not = p => {return x => !p(x);} ;
```

- Beispiele: Tests von Prädikaten und deren Kombination

```
In [30]: // test
         isBigger5(6)

Out[30]: true

In [31]: not(isOdd)(5)

Out[31]: false

In [32]: and(isOdd, isEven)(1,6)

Out[32]: true
```

Closures

Ein Closure ist eine Funktion mit zugeordnetem Speicherbereich. Auf den Speicher kann nur aus der Funktion heraus zugegrifen werden.

Closures sind ein wichtiges Hilfsmittel im Rahmen der funktionalen Programmierung. Dabei wird einer Funktion ein Kontext zugeordnet, auf den sie im Rahmen ihrer Abarbeitung lesend und schreibend zugreifen kann. Dadurch erhält die Funktion einen Zustand. Aus Sicht der objekt-orientierten Programmierung ist ein Closure ein Objekt mit genau einer Methode.

Beim schreibenden Zugriff auf die Closure-Variable ist darauf zu achten, dass im Rahmen der Funktionalen Programmierung, die Funktion trotzdem "Pure" bleiben muss.

- Beispiele: erlaubte Verwendung von Closures im Rahmen der FP

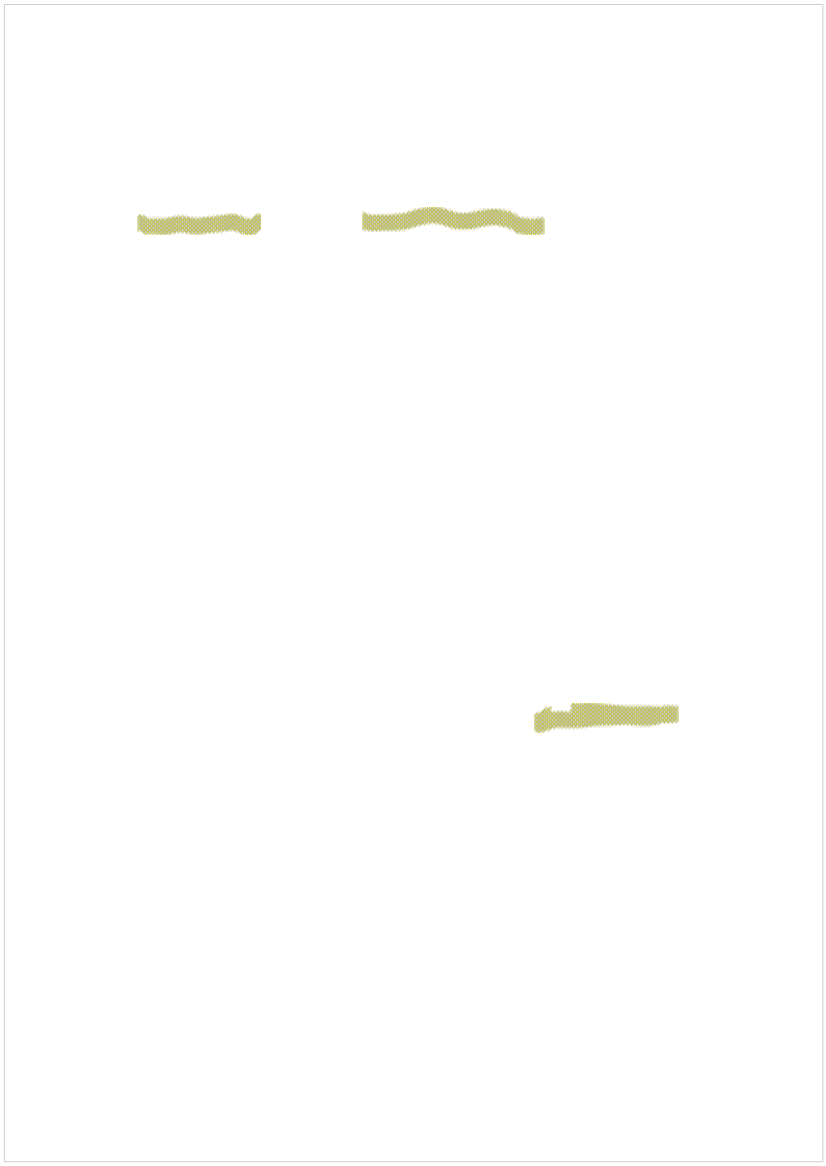
```
In [33]: // closure variables are in capital letters
         function multiplyWith(X) {       // function declaration
            const C = X;                  // closure scope
            return function(y) {
               return C * y;
            };
         }

         var times10 = multiplyWith(10);  // closure (function + scope)
         var times5 = multiplyWith(5);

In [34]: times5(4)

Out[34]: 20

In [35]: times10(4)

Out[35]: 40

In [36]: //closure with IIFE       immediately-invoked function expression
         var times3 = (function(X){        // function expression
            const C = X;                            // C: closure scope
            return function (y) {
               return C * y;
            };
         })(3);                                                 // immediate invocation

In [37]: times3(4);                                  // closure: times3(4)

Out[37]: 12
```

- Beispiel: "verbotene" Verwendung von Closures im Rahmen der FP

```
In [38]: /*
          * "bad" use of closures
          * this closure changes state: to be avoided in functional programming !!!
          */
         function greaterThan(X) {
            return function(y) {
               X++;                  // side effect!
               return y > X - 1;
            };
         }
         var gT10 = greaterThan(10); // closure

In [39]: gT10(11);

Out[39]: true

In [40]: gT10(11);

Out[40]: false
```

5.2.3 Transformation und Komposition von Funktionen

Das Umgestalten von Funktionen und das Kombinieren von einfachen Funktionen zu komplexen Anwendungen ist die Grundlage der Funktionalen Programmierung. Wenn die Bausteine entsprechend der Prinzipien der FP entwickelt werden, können sie im Anschluss unabhängig vom Kontext (von Server, Laufzeit, Zustand des Hauptspeichers, ...) zu verteil- und parallelisierbaren Anwendungen kombiniert werden. Dabei sind HOFs und Closures die wichtigsten Werkzeuge. Für die weitere Betrachtung definiern wir jedoch zuvor noch den Begriff *Stelligkeit* sowie eine spezielle Funktion - die *Identische Abbildung*.



```
                    return x + b + c;
                };
            }
            return x + b + c;
        };
    }
    return function (c) {
        return x + y + c;
    };
}
return x + y + z;
};
```

```
In [54]: //test
         add3(5,5,1);
```

```
Out[54]: 9
```

```
In [55]: add3(5,5)(1);
```

```
Out[55]: 9
```

```
In [56]: add3(5)(5,1);
```

```
Out[56]: 9
```

```
In [57]: add3(5)(5)(1);
```

```
Out[57]: 9
```

2. Als generische curry-Funktion

```
In [58]: // currying function for functions with arbitrary arity
         function curry(f) {
            return function curried(...args) {   // ...args expands array to argument list
               if (args.length >= f.length) {    // number of arguments is equal to arity(f) -> return
                  return f(...args);
               } else {
                  return function(...args2) {    // number of arguments is lower than arity(f) -> return
                     return curried(...args.concat(args2)); // fill ...args until arity(f) is reached
                  };
               }
            };
         };
```

```
In [59]: // functions to be curried
         function prod3(a, b, c) {
            return a * b * c;
         }
         function sum4(a, b, c, d) {
            return a + b + c + d;
         }
         function div3(a, b, c) {
            return a / b / c;
         }
```

```
In [60]: //test
         curry(prod3)(5,2,3);
```

```
Out[60]: 30

In [61]: curry(prod3)(5,2)(3);       // partially applied

Out[61]: 30

In [62]: curry(prod3)(5)(2)(3);      // fully curried

Out[62]: 30

In [63]: curry(sum4)(5,2)(3,4);  // partially applied

Out[63]: 14

In [64]: curry(sum4)(5)(2)(3)(4); // fully curried

Out[64]: 14

In [65]: curry(div3)(12)(2)(3);  // fully curried

Out[65]: 2
```

Die generische `curry`-Funktion kann auf Funktionen mit beliebiger Stelligkeit angewendet werde.

Komposition von Funktionen

- **Definition**: Eine Komposition aus zwei Funktionen ist folgendermaßen definiert

$$(g \circ f)(x) := g(f(x))$$

  d.h. die Abarbeitung erfogt von rechts nach links.

- Die Komposition von Funktionen muss folgende **Eigenschaften** erfüllen:

1. Der Type der Ausgabe der rechten Funktion muss gleich dem Eingabetyp der linken Funktion sein. Man schreibt mit Hughes Arrow-Syntax `f :: A -> B` und `g :: B -> C`.
2. Die Komposition ist assoziativ

$$h \circ (g \circ f) = (h \circ g) \circ f = h \circ g \circ f$$

3. Sei A der Typ des Eingabparameters von f und B der Typ des Eingabeparameters von g (und wg. 1. auch der Typ des Rückgabeparameters von f), dann gilt für die Verknüpfung mit der identischen Abbildung:

$$f \circ id_A = f$$

   und

$$id_B \circ f = f$$

- Die **Implementierung** der Komposition von zwei Funktionen lässt sich in JavaScript direkt aus der Definition ableiten:

```
In [66]: // define functions to be composed
         var add1 = x -> x+1;
         var square = x -> x*x;
         var stringResult = x -> "Result = "+x;
```

```
In [67]: // (f * g)(x) = f(g(x))
         var compose = function(f, g) {
            return function(x) {
               return f(g(x));
            };
         };

         // test
         compose(add1,square)(2);                  // same return types

Out[67]: 5

In [68]: compose(square,add1)(2)

Out[68]: 9

In [69]: compose(stringResult, add1)(2)  // different return types

Out[69]: 'Result = 3'

In [70]: compose(add1, stringResult)(2);

Out[70]: 'Result = 21'
```

- Kompositionen von beliebig vielen Funktionen können folgendermaßen implementiert werden:

```
In [71]: var multicompose = function(...args){                    // ...args stores arguments in array
            let funcs = args;                                     // array of functions - closure var.
            return function f(x){
               let current_f = funcs.pop();                       // get last function of array
               if (typeof current_f === 'function') {
                  return f(current_f(x));                         // f(g(h(...(x))))
               }
               else {
                  return x;                                      // last function reached
               }
            };
         };

In [72]: // test
         multicompose(stringResult,add1,square,square)(2);        // evaluate right to left

Out[72]: 'Result = 17'

In [73]: multicompose(stringResult,add1,add1,add1,add1,add1,square)(2);

Out[73]: 'Result = 9'
```

5.2.4 Laufzeiteigenschaften

Lazy Evaluation (Non-strict evaluation)

- **Definition**: Eine Funktion wird erst dann ausgewertet, wenn ihr Rückgabeparameter benötigt wird.
- **Vorteile**:
  - Verzicht auf Auswerung unnötiger Ausdrücke
  - Effiziente Memoisation (s.u.)
  - erlaubt Berechnung unendicher Datenstrukturen

Rekursion statt Iteration

- **Vorteile** gegenüber der Iteration:
  - keine "versteckte" Zustandsänderung (z.B. durch Laufvariable)
  - Optimierung durch Lazy Evaluation und Memoisation
- **Beispiel**: rekursive Implementierung der Fibonacci-Folge

```
In [74]: // straight forward...
         function fib(n) {
            if(n<-2) {
               return 1;
            } else {
               return fib(n-1) + fib(n-2);
             }
         }

In [75]: // test
         fib(7);

Out[75]: 13
```

- Problem: Nicht endständige Rekursion (s.u.) bewirkt langsame Verarbeitung und hohen Speicherverbrauch, da für jede Rekursionstiefe Daten gespeichert werden müssen.
- Implementierung als **Endrekursion**: Der darauffolgende Rekursionsaufruf ist die letzte Operation innerhalb einer rekursiven Funktion. Viele Compiler haben Möglichkeiten, endrekursive Funktionen hinsichtlich Laufzeit und Speicherverbrauch zu optimieren.
- **Beispiel**: endrekursive Imnlementierung der Fibonacci-Folge

```
In [76]: // tail call recursion (ECMAScript 6 offers tail call optimization)
         function fib_tcr(n, a, b) {
            switch (n) {
            case 1:
               return b;
            default:
               return fib_tcr(n - 1, b, a + b);
            }
         }
         // initialize with closure => currying
         function fib_iri(n) {
            return fib_tcr(n, 0, 1);
         }

In [77]: // test
         fib_iri(7);

Out[77]: 13
```

Der Performanzgewinn durch die Endrekursion lässt sich einfach messen:

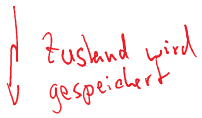
```
In [78]: var t = process.hrtime();
         fib(20);
         t = process.hrtime(t);
         t[1]/1000;                    //in us
```

```
Out[78]: 2157.075

In [79]: var t_tcr = process.hrtime();
         fib_tri(20);
         t_tcr = process.hrtime(t_tcr);
         t_tcr[1]/1000;                  //in us

Out[79]: 9.62

In [80]: (t[1]/t_tcr[1]).toFixed(1);     // gain

Out[80]: '224.2'
```

Memoisation

- Memoisation (eng. memoization) bietet neben der Endrekursion eine weitere Laufzeitoptimierung für Funktionen.
- Wegen der Referenziellen Transparenz liefern Funktionsaufrufe mit identischen Argumenten **immer** den selben Rückgabewert. Es genügt eine Funktion einmal auszuwerten und das Ergebnis zu cachen. Als Cache dient hier wieder eine Closure-Variable:

```
In [81]: /**
          * Memoisation using the [redacted] pattern
          * Remark: Memoisation is a sideeffect since
          * it modifies the state of a function.
          * It keeps referential transparency nevertheless.
          */

         // Memoisation
                 // requires referential transparency
                 // uses closure
         function memoize(f){
             var cache = {};
             return function memoized(x){
                 if (!cache[x]){
                         cache[x] = f(x);
                 }
                 return cache[x];
             };
         }
```



Der Performanzgewinn durch die Memoisation lässt sich auch hier messen:

```
In [82]: // measure without memoization
         var t = process.hrtime();
         fib(20);
         t = process.hrtime(t);
         t[1]/1000;                      //in us

Out[82]: 73.076

In [83]: // decorate
         var fib_mem = memoize(fib);
```

```
In [84]: //measure with memoization
         var tm = process.hrtime();
         fib_mem(20);
         tm = process.hrtime(tm);
         tm[1]/1000;                    //in us

Out[84]: 200.423

In [85]: (t[1]/tm[1]).toFixed(1);       // gain

Out[85]: '0.4'
```

Chapter 6

Patterns

6.1 Funktoren

6.1.1 Motivation

Gegeben sei folgende einstellige Funktion `double`.

```
In [86]: // define a function that multiplies a parameter
         var double = x -> 2*x;
         double(7);
```

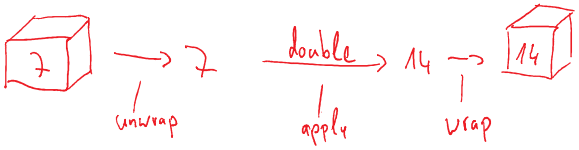
```
Out[86]: 14
```

Diese soll nun auf ein Array angewendet werden.

```
In [87]: // what happens if we apply double() to an array?
         double([2, 5, 7]);
```

```
Out[87]: NaN
```

Parameter tauchen in der Informatik häufig in unterschiedlichen Formen "verpackt" auf (in Objekten, als Arrays, ...). Häufig steht man vor der Aufgabe, dass auf diese Container Funkionen angewendet werden sollen. In diesem Fall muss der primitive Parameter aus der Verpackung entnommen, die Funktion angewandt und das Ergebnis wieder "verpackt" werden. Dies ist die Aufgabe eines Funktors.



6.1.2 Definitionen

Für das Konzept des Funktors existieren verschiedene Definitionen. Wir betrachten hier zunächst die mathematische Definition aus der Kategorientheorie.

- **Definition** (*math.*): Ein Funktor ist eine strukturerhaltende (= homomorphe) Abbildung zwischen Kategorien:

  Für zwei gegebene Kategorien **C** und **D** gibt es einen Funktor

  $$F : \mathbf{C} \to \mathbf{D},$$

  so das gilt:

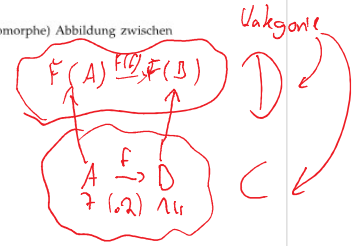
  1. Für jedes Objekt $A \in \mathbf{C}$ existiert ein Objekt $F(A) \in \mathbf{D}$,
  2. Für jede Abbildung

     $$f : A \to B \in \mathbf{C}$$

     gibt es eine Abbildung

     $$F(f) : F(A) \to F(B)$$

     in **D**.



In der Kategorientheorie verwendet man häufig *kommutative Diagramme* zur Darstellung dieser Abhängigkeiten.

- **Definition** (*inf.*): In der Informatik wird ein Funktor als Objekt definiert, das eine Funktion zur Erzeugung des Objekts (`of`-Funktion) sowie eine strukturerhaltende Abbildung (`map`-Funktion) implementiert. Häufig wird auch `map`- oder `fmap`-Funktion selbst als Funktor bezeichnet.

Anstatt der mathematischen Notation mit Klammern verwendet man in der Informatik eher die Arrow-Syntax nach John Hughes:

- `f`-Funktion: `f :: a -> b`
- `of`-Funktion: `F :: a -> F a`
- `fmap`-Funktion: `F f :: F a -> F b`.

6.1.3 Eigenschaften

Gegeben seine seien zwei beliebige Funktionen f und g, sowie die identischen Abbildung id mit $id : x \to x$.
Ein Funktor F muss folgende Eigenschaften erfüllen:

1. für jedes $A \in \mathbf{C}$ gilt:
$$F(id_A) = id_{F(A)}$$

2. für zwei Funktionen f und g in $\mathbf{C}$ gilt:
$$F(g \circ f) = F(g) \circ F(f)$$

6.1.4 Ein Container-Funktor

Folgendes Coding repräsentiert einen Funktor, der eine Variable `value` enthält und eine `zap`-Funktion implementiert. Die `of`-Funktion ist der Funktionsaufruf selbst. Die `extract`-Methode ist nicht Teil eines Funktors und wird hier nur für die Ausgabe benötigt.

```
In [88]: function Container(val) {          // F :: a -> F a : of()-function

           var value = val;

           // "public" functions
           return {

                zap : function(f) {          // F f :: F a -> F b
                    return Container(f(value));
                },

                // Not part of Functor! Just for logging
                extract : function() {          // F a -> a,
                        return value;
                }
           };
         }
```

- **Test**: Wegen der schwachen Typisierung kann der Container beliebige Typen von `value` beinhalten.

```
In [89]: var toUpper = x => x.toUpperCase();    // Functions to be mapped
         var trim = x => x.trim();

In [90]: var cont = Container("      Hello World    ");
         cont.map(trim).map(toUpper).extract();

Out[90]: 'HELLO WORLD'

In [91]: var double = x -> 2*x;                 // Functions to be mapped
         var minus5 = x -> x   5;

In [92]: var cont = Container(7);
         cont.map(minus5).zap(double).extract();

Out[92]: 4
```

Wir müssen noch überprüfen, ob der Container-Funktor auch die Eigenschaften $F(id_A) = id_{F(A)}$ sowie $F(g \circ f) = F(g) \circ F(f)$ erfüllt. Hierzu benötigen wir wieder die `id` und `compose`-Funktion.

```
In [93]: var id = function(x){                  // id(x) = x  | Identity function
                return x;
         };

         var compose = function(f, g) {          // (f . g)(x) = f(g(x))
             return function(x) {
                 return f(g(x));
             };
         };
```

1. $F(id_A) = id_{F(A)}$

```
In [94]: Container(id(5)).extract();   //F(id_A)

Out[94]: 5

In [95]: id(Container(5)).extract();   //id_F(A)

Out[95]: 5
```

2. $F(g \circ f) = F(g) \circ F(f)$

```
In [96]: var F = Container(5);

In [97]: F.map(compose(double, minus5)).extract();

Out[97]: 8

In [98]: F.map(minus5).map(double).extract();

Out[98]: 8
```

6.1.5 Ein Array-Funktor

Arrays sind eine der wichtigsten Anwendungsbereiche von Funktoren. Hier ist der Nutzen einer `map`-Methode offensichtlich. Folgendes Beispiel soll das verdeutlichen.

```
In [99]: /**
          * Array FunctorJoachim Orb, 2017
          */

         function ArrayFunctor(initArray){                  // This is a Functor !!!

           var currentArray = initArray;

           // private functions
           function pMap(fn, oldArr, newArr) {
             var arr = oldArr.slice(0);       // deep copy to preserve state (shift() alters Array!)
             if (arr.length === 0) {
               return newArr;                                 // start with newArr as initial array (now
             } else {
               var first = arr.shift();                  // get first element and reduce current array b
               newArr.push(fn(first));                  // add mapped element to newArray if fn returns t
               return pMap(fn, arr, newArr);          // dives down filter with n-1
             }
           }

           // "public" functions
           return {

             map : function(f) {                              // F f :: F a -> F b
               return ArrayFunctor(pMap(f, currentArray, []));
             },

             of : function() {                              // F :: a -> F a
               return ArrayFunctor([].slice.call(arguments));
             },
```

```
    // Not part of Functor! Just for logging
    extract : function() {                  // F a -> a,
      return currentArray.toString();
    }
  };
}
```

- **Anmerkung**: Der `ArrayFunctor` ist hier ein Funktor eines Funktors, da Variablen vom Typ `Array` in JavaScript bereits als Funktoren implementiert sind.

```
In [100]: [1,3,5,7,11].map(x => x * 2);

Out[100]: [ 2, 6, 10, 14, 22 ]
```

- **Funktionaler Test**: Hier soll aber der `ArrayFunctor` getestet werden.

```
In [101]: // Test Functor
          // define some functions for map
          var times2 = x -> 2*x;
          var minus5 = x -> x - 5;

In [102]: // execute
          var arrayChain = ArrayFunctor();
          arrayChain.of(1,3,5,7)
                    .map(times2)
                    .map(minus5)
                    .extract();

Out[102]: '-3,1,5,9'
```

- **Überprüfung**: Auch hier wollen wir die beiden notwendigen Eigenschaften eines Funktors überprüfen. Wir verwenden wieder die `id` und `compose`-Funktionen.

1. $F(id_A) = id_{F(A)}$

```
In [103]: ArrayFunctor(id([1,3,5,7])).extract();      //F(id_A)

Out[103]: '1,3,5,7'

In [104]: id(ArrayFunctor([1,3,5,7])).extract();      //id_F(A)

Out[104]: '1,3,5,7'
```

2. $F(g \circ f) = F(g) \circ F(f)$

```
In [105]: var F = ArrayFunctor().of(1,3,5,7);

In [106]: F.map(compose(times2, minus5)).extract();

Out[106]: '-3,-1,3,7'

In [107]: F.map(minus5).map(times2).extract();

Out[107]: '-3,-1,3,7'
```

6.1.6 Ein `Maybe`-Funktor

Ein typischer Anwendungsfall für Funktoren und Monaden (siehe nächstes Kapitel) ist die robuste Berechnung von Ergebnissen bei unsicheren Funktionsparametern. Wegen der referenziellen Transparenz müssen diese immer ein eindeutiges Ergebnis liefern und dürfen keine Wirkungen (z.B. log-Einträge) haben.

```
In [108]: function MaybeFiniteFunctor(val) {          // F :: a -> F a : of()-function

            var value = val;

            // "public" functions
            return {
              map : function(f) {          // F f :: F a -> F b
                return (isFinite(f(value))) ? MaybeFiniteFunctor(f(value)) : MaybeFiniteFunctor(null);
              },

              // Not part of Functor! Just for logging
              extract : function() {     // F a -> a,
                return value;
              }
            };
          }
```

- **Funktionaler Test**:

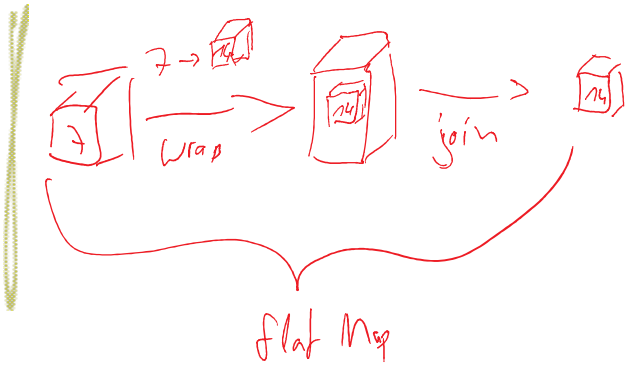
```
In [109]: var divide10By = x => 10/x;

In [110]: MaybeFiniteFunctor(5).map(divide10By).extract();

Out[110]: 2

In [111]: MaybeFiniteFunctor(0).map(divide10By).extract();

Out[111]: null
```

- **Überprüfung**: Auch hier können wir die Funktor-Regeln überprüfen:

```
In [112]: MaybeFiniteFunctor(id(0)).extract();   //F(id_A)

Out[112]: 0

In [113]: id(MaybeFiniteFunctor(0)).extract();   //id_F(A)

Out[113]: 0

In [114]: MaybeFiniteFunctor(5).map(compose(divide10By, minus5)).extract();

Out[114]: null

In [115]: MaybeFiniteFunctor(5).map(minus5).map(divide10By).extract();

Out[115]: null
```

6.2 Monaden

> *"Once you understand monads, you immediately become incapable of explaining them to anyone else"*
> Gilad Bracha



6.2.1 Definition

Eine Monade ist ein abstrakter Datentyp, der Operationen auf einen zugrundeliegenden "einfacheren" Typ anwendet, und aus den Ergebnissen eines "höheren" Typs, eine neue Instanz des abstrakten Datentyps erstellt.

6.2.2 Motivation

Bei den **Funktoren** ging es darum, die Funktion eines einfachen Typs
`f :: a -> b`
auf einen Container `F a` anzuwenden. Bei konkreten Problemen steht man jedoch häufig vor der Herausforderung, dass die Basisfunktion selbst einen Funktor erzeugt, also:
`g :: a -> F b`.
Nach Anwendung der `fmap`-Funktion ergäbe sich dann
`F g :: F a -> F F b`.
Um das gewünschte Ergebnis zu erreichen, muss `F F b` zu `M b` "abgeflacht" (flattened) werden. Mappen und "Abflachen" sind die wichtigsten Aufgaben einer Monade.

6.2.3 Eigenschaften

Eine Monade muss mindestens die beiden folgenden Operationen implementieren:

- `unit / return :: a -> M a` : Konstruktion der Monade
- `flatMap / bind :: (a -> M b) -> M a -> M b` : Map-Funktion mit "Abflachung"

Häufig werden zusätzlich noch folgende Funktionen bereitgestellt:

- `fmap / fmap :: (a -> b) -> M a -> M b` : siehe Funktor

- `join` / `flat` :: M (M a) -> M a : "Abflachen" der Monade
- `lift` :: (c -> b) -> (a -> M b) : wendet eine Funktion an und erzeugt monadischen Typ

Bemerkung:
`flatMap = join.map`

6.2.4 Eine Array-Monade

Bei unserem Array-Funktor wurden nur einfache Funktionen des Typs `f :: c -> b` gemappt. Die folgende Funktion `primeFactors` hat aber selbst einen Rückgabewert vom Typ Array, also ist `f :: a -> F b`.

```
In [116]: var primeFactors = function(num){
            var root = Math.sqrt(num),
            result = arguments[1] || [],    // get unnamed parameter from recursive calls
            x = arguments[2] || 2;

            if(x === 2 && num % x){         // if not divisible by 2
              x = 3;                        // assign first odd
            }
            while((num % x) && ((x = x + 2) < root)){}        // iterate odds
            // if no factor found then num is prime
            x = (x <= root) ? x : num;
            result.push(x);   // push latest prime factor

            // if num isn't prime factor make recursive call
            return (x === num) ? result : primeFactors(num/x, result, x) ;
          };
```

- **Beispiel**:

```
In [117]: // test
          primeFactors(18);

Out[117]: [ 2, 3, 3 ]
```

Wird `primeFactors` mit `map` auf ein Array angewendet entsteht ein Array von Arrays, da `F f :: F a -> F F b`. Anstatt der `map`- benötigen wir `flatMap`-Funktion.

```
In [118]: /**
           * Array Monad Joachim Orb, 2018
           */
          // Monad is an abstract type
          function MonadArray(initArray) {

            var arr = initArray.slice(0);
            //arr = [];

            // private functions

            function pUnit(initArray){
              return initArray.slice(0);
            }

            function pMap(fn, oldArr, newArr) {                              // private
```

```
  var array = oldArr.slice(0);                                  // deep copy to preserve stat

  if (array.length === 0) {
    return newArr;                                                        // start with new
  } else {
    var first = array.shift();                                  // get first element and red
    newArr.push(fn(first));                                       // add mapped element to
    return pMap(fn, array, newArr);                           // dives down filter with n-1
  }
}

function pFlat(oldArr, newArr){

  if (Array.isArray(oldArr)) {

    var array = oldArr.slice(0);                              // deep copy to preserve state (shi

    if (array.length === 0){
      return newArr;
    }
    else {
      var first = array.shift();                              // cut array in two parts: first me
      pFlat(first,newArr);                                      // handle first
      return pFlat(array,newArr);                               // handle rest
    }
  } else {
    newArr.push(oldArr);
    return newArr;
  }
}

// "public" functions
return {

  unit : function() {                                                                  // A M
    var initArr = Array.prototype.slice.call(arguments);        // unit :: a -> Ma
    //console.log(initArr);
    return MonadArray(initArr);
  },

  lift : function(fn) {                                                          // lift :: (
    return function(x){
      return MonadArray([fn(x)]);
    }
  },

  map : function(fn) {                                                         // Functor ma
    return MonadArray(pMap(fn, arr, []));                     // map :: ( a -> b ) -> Fa
  },

  flat : function() {                                                          //
    return MonadArray(pFlat(arr, []));
  },
```

```
    flatMap : function(fn) {                                                    // λ Mona
      return MonadArray(pFlatMap(fn, arr, []), []);          // Ma -> ( a -> Mb ) -> Mb
    },

    extract: function() {
      return arr;
    }

  };
}
```

In [119]:
```
// some functions for testing
var square = x -> x*x;
var isEven = x => (x%2) === 0;
```

In [120]:
```
// test lift
MonadArray([]).lift(square)(5).extract();
```

Out[120]:
```
[ 25 ]
```

In [121]:
```
// test functor
MonadArray([]).unit(2,4,7).map(square).map(isEven).extract();          // Functors
```

Out[121]:
```
[ true, true, false ]
```

Wird `primeFactors` mit `map` auf ein Array angewendet entsteht ein Array von Arrays, da `F f :: F a -> F F b`.

In [122]:
```
// test map primeFactors
MonadArray([]).unit(1,2,3,4,5,6,7,8,9,10).map(primeFactors).extract();
```

Out[122]:
```
[ [ 1 ],
  [ 2 ],
  [ 3 ],
  [ 2, 2 ],
  [ 5 ],
  [ 2, 3 ],
  [ 7 ],
  [ 2, 2, 2 ],
  [ 3, 3 ],
  [ 2, 5 ] ]
```

Wenn darauf erneut Funktionen angewendet werden sollen, kommt es zu unerwünschtem Verhalten.

In [123]:
```
// test map on "deep structure": does not work!
MonadArray([]).unit(1,2,3,4,5,6,7,8,9,10).map(primeFactors).map(square).extract();
```

Out[123]:
```
[ 1, 4, 9, NaN, 25, NaN, 49, NaN, NaN, NaN ]
```

Anstatt der `map`- benötigen wir `flatMap`-Funktion, welche wegen `(a -> M b) -> M a -> M b` eine Monade gleichen Typs erzeugt.

In [124]:
```
// test flatMap: works
MonadArray([]).unit(1,2,3,4,5,6,7,8,9,10).flatMap(primeFactors).flatMap(square).extract();
```

Out[124]:
```
[ 1, 4, 9, 4, 4, 25, 4, 9, 49, 4, 4, 4, 9, 9, 4, 25 ]
```

Das selbe Ergebnis kann auch durch eine Verkettung von `map` und `flat`-Funktion erreicht werden.

```
In [125]: // test flat
          MonadArray([]).unit([1,[2,3],4],5,6,[7,8]],9,[[[10]]]).flat().extract();

Out[125]: [ 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 ]

In [126]: // test first map then flat
          MonadArray([]).unit(1,2,3,4,5,6,7,8,9,10).map(primeFactors).flat().map(square).extract();
          // flatMap() = map().flat()

Out[126]: [ 1, 4, 9, 4, 4, 25, 4, 9, 49, 4, 4, 4, 9, 9, 4, 25 ]
```

6.2.5 Eine Writer-Monade

Problem

Monaden werden in der Funktionalen Programmierung häufig eingesetzt, um verbotene Seiteneffekte von Funktionen zu umgehen oder um Kontextinformationen an/von Funktionen zu übergeben. Ein typisches Beispiel für Seiteneffekte sind Einträge in einen Log. Aufgrund der Referenziellen Transparenz sind diese in der FP nicht erlaubt bzw. je nach Programmiersprache nicht möglich.

Lösung

Rückgabeparameter werden zusammen mit den Log-Einträgen in einen Container verpackt und von Funktionsaufrauf zu Funktionsaufruf weitergegeben. Da die Funktionen selbst Werte *und* Log-Einträge produzieren (`a -> M b`) benötigen wir eine Monade.

Implementierung

Zunächst benötigen wir Funktionen, die Rückgabewerte und Log-Einträge in einem Array zurückliefern

```
In [127]: // basic functions
          // without logging
          var square = function(x) {
                  return x*x;
          };
          // with logging
          var square_log = function(x) {
                  return [x*x, 'square_log'];
          };

          var minus5_log = function(x) {
                  return [x - 5, 'minus5_log'];
          };

          var isEven_log = x => [(x%2) === 0, 'isEven_log'];
```

Die Implementierung der Monade kann nach unten stehendem Beispiel erfolgen.

```
In [128]: //helper
          //(f . g)(x) = f(g(x))
          var compose = function(f, g) {
```

```
    // return function(x) {
    //     return f(g(x));
    // };
};
```

In [129]:
```
/**
 * Writer monad Joachim Orb, 2016
 */

// Monad is an abstract type
function MonadWriter(initArray) {

  var arr = initArray.slice(0);

        // private functions
  function pFlat(oldArr, newArr){

    if (Array.isArray(oldArr)) {

       var array = oldArr.slice(0);        // deep copy to preserve state (shift() alters Arr

       if (array.length === 0){
         return newArr;
       }
       else {
         var first = array.shift();          // cut array in two parts: first member and rest
         pFlat(first,newArr);                  // handle first
         return pFlat(array,newArr);          // handle rest
       }
    } else {
       newArr.push(oldArr);
       return newArr;
    }
  }

  // "public" functions
  return {

    unit : function(x) {                          // A Monad is a type that implements unit!
       return MonadWriter([x,'unit(' + x +')']);        // a -> Ma
    },

    lift : function(fn) {                                //lift :: (a -> b)-> (a -> Mb)
       return function(x){
         return MonadWriter([fn(x), 'lifted_'+fn.name]);
       };
    },

    map : function(fn) {                                                  // Functor
       return MonadWriter([fn(arr[0]), arr[1]]);
    },

    flat : function() {                                                        //  M (M a) ->
       var flattenedArr = pFlat(arr, []);
```

```
        return MonadWriter([flattendArr.shift(), flattendArr.reverse().join()]);
      },

      flatMap : function(fn) {                                          // A Monad is a type that
        return MonadWriter([fn(arr[0])[0], arr[1] +',' + fn(arr[0])[1]]);       //
      },

      extract : function() {
        return arr;                          // immutable by nature; but side effect!
      }

    };
  }
```

Tests

- `lift :: (a -> b) -> (a -> M b)`

```
In [130]: // test lift
          MonadWriter([]).lift(square)(5).extract();
```

```
Out[130]: [ 25, 'lifted_square' ]
```

- Mit `unit` kann ein monadischer Typ erzeugt werden.

```
In [131]: // test map
          MonadWriter([]).unit(3).extract();
```

```
Out[131]: [ 3, 'unit(3)' ]
```

- Auch hier kann **nicht** die `map`-Funktion angewendet werden.

```
In [132]: // test map
          MonadWriter([]).unit(3).map(square_log).extract();
```

```
Out[132]: [ [ 9, 'square_log' ], 'unit(3)' ]
```

```
In [133]: //map() does not work!!!
          MonadWriter([]).unit(3).map(square_log).map(minus5_log).extract();
```

```
Out[133]: [ [ NaN, 'minus5_log' ], 'unit(3)' ]
```

- Mit Hilfe einer spezifischen `flat`-Funktion kann die Monade der Monade "abgeflacht werden (`flat :: M (M a) -> M a)`

```
In [134]: // test flat
          MonadWriter([]).unit(3).map(square_log).flat().errrcct();
```

```
Out[134]: [ 9, 'unit(3),square_log' ]
```

```
In [135]: MonadWriter([]).unit(3)
                                  .map(square_log).flat()
                                  .map(minus5_log).flat()
                                  .map(square_log).flat()
                                  .map(isEven_log).flat()
                                  .extract();
```

```
Out[135]: [ true, 'unit(3),square_log,minus5_log,square_log,isEven_log' ]
```

- Das selbe Ergebnis liefert auch die `flatMap`-Funktion der Monade.

```
In [136]: // test flatMap()
          MonadWriter([]).unit(3)
                                        .flatMap(square_log)
                                        .flatMap(minus5_log)
                                        .flatMap(square_log)
                                        .flatMap(isEven_log)
                                        .extract();
```

```
Out[136]: [ true, 'unit(3),square_log,minus5_log,square_log,isEven_log' ]
```

6.2.6 Weitere Monaden-Typen

Die Reader-Monade

Die Reader-Monade wird angewendet, wenn Kontextinformation an eine zu mappende Funktion übergeben werden soll, wenn also die Funktion vom Typ `f :: M a -> b` sein soll.

Die State-Monade

Die State-Monade wird verwendet, wenn Satusinformation zwischen den zu mappenden Funktionen weitergereicht werden soll, wenn also die Funktionen vom Typ `f :: M a -> M b` sein sollen. Man schreibt hier auch `f :: a -> (a,a)`.

Bibliography

[Ara17] Anto Aravinth. *Functional Programming with JavaScript Using EcmaScript 6*. Apress. 2017

[Fog13] Michael Fogus. *Functional JavaScript*. O'Reilly. 2013

[Mil14] Bartosz Milewski. *Category Theory for Programmers*. https://bartoszmilewski.com/2014/10/28/category-theory-for-programmers-the-preface/. posted 2014