Konzepte Funktionaler Programmierung in anderen Sprachen

Von Matthias Dieter Wallnöfer

Einstieg

- Die Welt der *Funktionalen Programmierung* anhand von Ocaml besprochen
- Großteil der Programmiersprachen die zur Zeit genutzt werden **prozedural** bzw. **objektorientiert**
  - Seit den letzten zehn-fünfzehn Jahren werden sie vermehrt mit **funktionalen Konzepten** ausgestattet
  - Ermöglicht eine zunehmende Nutzung dieser Aspekte neben Bewährtem
  - Pure Funktionen eignen sich für eine seiteneffektfreie Parallelisierung von Algorithmen, heutzutage arbeiten wir mit Multi-Core bzw. Verteilten Systemen

Vorausgesetzte Kriterien

- Ausnahmen („*Exceptions*“)
- Speicherbereinigung („*Garbage collection*“)

Betrachtete Kriterien

- Funktionen erster Klasse („*first-class functions*“)
- Partielle Auswertung („*Currying*“)
- Rekursion
- Typinferenz („*Type inference*“)
- Listen und Tupel
- Endrekursionsoptimierung („*Tail recursion*“)
- Musterabgleich („*Pattern matching*“)
- Unveränderliche Variablen in Hinblick auf Closures („*Referential transparency*“)

Betrachtete Programmiersprachen

- JavaScript/ECMAScript
- Python
- Java
- C#
- PHP
- C++ (und C)

JavaScript/ECMAScript

Funktionen erster Klasse

Traditionell

```
let x = function(f) { return f; }
```

Rückgabe: `x = function(f)`

```
x (function() { return 'Hallo Welt' })
```

Rückgabe: `function()`

```
x (function() { return 'Hallo Welt' }) ()
```

Rückgabe: `,Hallo Welt‘`

Ab ECMAScript 2015

```
let x = f => f
```

Rückgabe: `x = function(f)`

```
x (() => 'Hallo Welt') ()
```

Rückgabe: `,Hallo Welt‘`

Rekursion

```
let fak = n => n > 1 ? n * fak(n-1) : 1
fak(4)
```

Rückgabe: `24`

Partielle Auswertung

Achtung, so nicht (Interpreter definiert ‚b‘ einfach als Zahl die NaN ist):

```
let sum = (a,b) => a + b
sum (3)
```

Rückgabe: `NaN`

Aber:

```
let sum = a => (b => a + b)
sum (3)
```

Rückgabe: `function(b)`

```
sum (3) (3)
```

Rückgabe: `6`

Typinferenz

```
let x = 1
typeof(x)
```

Rückgabe: “number"

```
let x = "Hallo Welt"
typeof(x)
```

Rückgabe: „string"

Allerdings führt JS durch seine **schwache und dynamische Typisierung** auch oft **ungewollte Konversionen** aus → Abhilfe schafft TypeScript

Listen und Tupel

Listen können durch Arrays gut dargestellt werden:

- cons (::) → Beim Erzeugen `Array.unshift()`, beim Matchen `Array.shift()`
- List.append / @ → `Array.concat()`
- List.map → `Array.map()`

Tupel wie in Ocaml gesehen gibt es in JS derzeit noch nicht, ein Vorschlag ist in Ausarbeitung: https://github.com/tc39/proposal-record-tuple. Einstweilen lassen sie sich über Arrays oder Maps simulieren:

- Als Feld:
  ```
  let person = ['name','surname',50]
  person[0], person[1], person[2]
  ```
- Als Mappe:
  ```
  let myMap = new Map();
  myMap.set(0, 'zero');
  myMap.set(1, 'one');
  ```

Quellen: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array, https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Map

Endrekursionsoptimierung

Ab ECMAScript 2015 möglich: https://bdadam.com/blog/video-douglas-crockford-about-the-new-good-parts.html

~~Musterabgleich~~

Nicht vorhanden, nur über Umwege möglich: https://www.bramstein.com/writing/pattern-matching.html. Es gibt aber einen Vorschlag, der es in eine kommende Version schaffen könnte: https://github.com/tc39/proposal-pattern-matching

Unveränderliche Variablen

Standardmäßig kennt JS nur die in den iterativen Sprachen benutzten veränderlichen Variablen. Beispiel:

```
let x = 3
let f = () => x
x = 4
f()
```

Rückgabe: `4`

Der Trick besteht in der Einführung eines Default-Arguments mittels der Hilfsfunktion `bind()`:

```
let x = 3
let f = (x => x).bind(this, x)
x = 4
f()
```

Rückgabe: `3`

Quelle: https://developer.mozilla.org/de/docs/Web/JavaScript/Reference/Global_Objects/Function/bind

Python

Funktionen erster Klasse

Mittels lambda (= einzeilige Funktionen)

```
x = (lambda f: f)
x
```

Rückgabe: `<function <lambda> at ...>`

```
x (lambda: 'Hallo Welt')
```

Rückgabe: `<function <lambda> at ...>`

```
x (lambda: 'Hallo Welt') ()
```

Rückgabe: `'Hallo Welt'`

Normale Funktionen

```
def x(f):
        return f
x
```

Rückgabe: `<function x at ...>`

```
def f():
        return 'Hallo Welt'
x(f)
```

Rückgabe: `<function f at ...>`

```
x(f)()
```

Rückgabe: `'Hallo Welt'`

Rekursion

```
fak = lambda n: n * fak(n-1) if n>1 else 1
fak(4)
```

Rückgabe: `24`

Partielle Auswertung

Ähnlich wie bei JS nur mit spezieller Syntax möglich, ansonsten kommt es zu Fehlermeldung:

```
sum = lambda a: (lambda b: a + b)
sum
```

Rückgabe: `<function <lambda> at ...>`

`sum(3)`

Rückgabe: `<function <lambda>.<locals>.<lambda> at ...>`

```
sum(3)(4)
```

Rückgabe: `7`

Typinferenz

```
x = 7
type(x)
```

Rückgabe: `<class 'int'>`

```
x = "Hallo Welt"
type(x)
```

Rückgabe: `<class 'str'>`

Python ist wie Ocaml **stark getypt** und benötigt grundsätzlich Konvertieroperatoren, jedoch wird der Typ **dynamisch** festgestellt (https://docs.python.org/3/reference/datamodel.html). Im Gegensatz zu Ocaml kann eine Ganzzahl automatisch in eine Fließkommazahl überführt werden, d.h. `3 + 3.5` ist problemlos möglich.

Listen und Tupel

Listen werden wie folgt benutzt:

```
l = [1, 2, 3]
type(l)
```

```
Rückgabe: <class 'list'>
```

Die Funktionen und Operatoren heißen in Python aber völlig anders:
https://docs.python.org/3/tutorial/datastructures.html#more-on-lists

Tupel gibt es als eigenen Typ:

```
x = ('name', 'surname', 30)
type(x)
```

Rückgabe: `<class 'tuple'>`

```
x[0], x[1], x[2]
```

Endrekursionsoptimierung

Von Sprachendesignern explizit abgelehnt, daher nur als Erweiterung:
https://github.com/baruchel/tco

Musterabgleich

Nicht völlig vergleichbar zu Ocaml, Beispiel Fibonacci in Wikipedia (https://en.wikipedia.org/wiki/Functional_programming#Python):

```
fibonacci = (lambda n, first=0, second=1:
    [] if n == 0 else
    [first] + fibonacci(n - 1, second, first + second))
print(fibonacci(10))
```

Alternative: https://github.com/grantjenks/python-pattern-matching

Unveränderliche Variablen

Standardmäßig sind auch in Python die Variablen veränderlich. Man kann ein Default-Argument einführen, welches nur bei Definition der Funktion ausgewertet wird, um das funktionale Verhalten zu erreichen. Im folgenden Beispiel setzen wir `x=x`:

```
x = 3
f = lambda x=x: x
x = 4
f()
```

Rückgabe: `3`

Java

Funktionen erster Klasse

Bis Java 8 gab es nur die Objekte erster Klasse, dann kamen die Lambda-Objekte hinzu, die diese Fähigkeit anhand von versteckten anonymen Klassen nachrüstet. Hier das gewohnte Beispiel:

```
import java.util.function.Function;
import java.util.function.Supplier;

Function<Supplier<?>,Supplier<?>> x = f -> f;
System.out.println(x.apply(() -> "Hallo Welt").get());
```

Java **unterscheidet genau zwischen verschiedenen Funktionstypen**, d.h. Schnittstelle `Supplier` falls kein Eingabeparameter und Schnittstelle `Function` wenn Eingabe- und Ausgabeparameter erwünscht. Mehr dazu unter:
https://docs.oracle.com/javase/tutorial/java/javaOO/lambdaexpressions.html

Auch die Kombination mit separaten Methoden ist möglich (gleich wie in C#).

```
public static String func() {
    return "Hallo Welt";
}
...
Function<Supplier<?>,Supplier<?>> x = f -> f;
System.out.println(x.apply(MyClass::func).get());
```

Rekursion

Mit normalen Funktionen kein Problem, aber bei Lambdas ist es nicht so einfach, da man sich nicht selbst referenzieren kann:

```
Function<Integer, Integer> fak = n -> n > 1 ? n * fak.apply(n-1)
: 1; // übersetzt nicht
```

Mögliche Abhilfen sind auf Stack-Overflow beschrieben:
https://stackoverflow.com/questions/19429667/implement-recursive-lambda-function-using-java-8

Partielle Auswertung

Ab Java 8 mittels expliziter Lambda-Syntax möglich:

```
import java.util.function.Function;

Function<Integer, Function<Integer, Integer>> sum = a -> (b -> a
+ b);
Function<Integer,Integer> sum_ = sum.apply(3);
System.out.println(sum_.apply(3));
```

Typinferenz

Ab Java 10 ist die Typinferenz für lokale Variablen möglich (gleich wie in C#):

```
var x = 10;
```

Danach kann `x` im selben Scope nicht mehr neu definiert bzw. der Typ nicht geändert werden und Konversionen müssen in weniger Fällen explizit durchgeführt werden (bspw. `int` in `float`/`double`/`long`/`String`) als es in Ocaml der Fall ist.

Java ist darum etwas weniger **stark** als Ocaml getypt, die Typinferenz erfolgt bei der Übersetzung, was die Typisierung **statisch** belässt.

Listen und Tupel

Was Listen betrifft, existieren in Java die von der Schnittstelle `java.util.List` abgeleiteten Klassen (`ArrayList`, `LinkedList`), die aber eine völlig andere Schnittstelle wie jene von Ocaml aufweisen.

Tupel gab es lange Zeit nicht, aber ab Java 16 haben sich „*Records*“ als Alternative etabliert: https://openjdk.org/jeps/395: `record Point(int x, int y) { }`

~~Endrekursionsoptimierung~~

Gewisse JVM-Sprachen wie Scala unterstützen sie, Java aber bislang nicht.

Quellen: https://softwareengineering.stackexchange.com/questions/272061/why-doesnt-java-have-optimization-for-tail-recursion-at-all, https://softwareengineering.stackexchange.com/questions/157684/what-limitations-does-the-jvm-impose-on-tail-call-optimization

Musterabgleich

Pattern-Matching, bereits längere Zeit im Gespräch (siehe https://cr.openjdk.java.net/~briangoetz/amber/pattern-match.html), ist ab Java 21 in ähnlichem Ausmaße von C# implementiert: https://openjdk.org/jeps/406.

```
static String formatterPatternSwitch(Object o) {
    return switch (o) {
        case Integer i -> String.format("int %d", i);
        case Long l    -> String.format("long %d", l);
        case Double d  -> String.format("double %f", d);
        case String s  -> String.format("String %s", s);
        default        -> o.toString();
    };
}
```

Oder bei einer selbstdefinierten Liste:

```
static record MyList<E>(E head, MyList<E> tail) { };

static <E> void iter(java.util.function.Consumer<E> f, MyList<E>
l) {
```

```
        switch(l) {
        case MyList(var h,var t): f.accept(h); iter(f,t); break;
        case null: break;
        }
    }

    public static void main(String args[]) {
        var list = new MyList<>(1, new MyList<>(2, new MyList<>(3,
    null)));
        iter(System.out::println, list);
    }
```

Unveränderliche Variablen

Man kann Variablen als `final` deklarieren, so dass sie einer Lambda-Funktion zur Verfügung steht. Änderungen sind dann aber nicht mehr erlaubt (im Endeffekt gleich zu C#):

```
final int x = 3;
Supplier<Integer> f = () -> x;
//x = 4; // nicht mehr möglich
System.out.println(f.get());
```

C#

Funktionen erster Klasse

sind in C# mithilfe der Delegates (auch Lambdas genannt) realisierbar. Deren Umsetzung unterscheidet sich nicht wesentlich von Java, wie das folgende Beispiel zeigt:

```
Func<Func<string>,Func<string>> x = f => f;
Console.WriteLine(x(() => "Hallo Welt")());
```

Man sieht, dass die **beabsichtigten Datentypen genau zu spezifizieren** sind.

Die Delegates stehen aber auch den herkömmlich definierten Methoden offen (gleich wie in Java):

```
public static string func() {
    return "Hallo Welt";
}
...
Func<Func<string>,Func<string>> x = f => f;
Console.WriteLine(x(func)());
```

Quelle: https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/statements-expressions-operators/lambda-expressions

Rekursion

Dieselbe Situation wie bei Java auch in C#, `fak` kann nicht aufgelöst werden:

```
Func<int, int> fak = n => n > 1 ? n * fak(n-1) : 1; // übersetzt nicht
```

Der Workaround ist allerdings einfacher, es reicht die Funktion zuerst mit einem Dummy (`null`) zu definieren:

```
Func<int, int> fak = null;
fak = n => n > 1 ? n * fak(n-1) : 1;
Console.WriteLine(fak(4));
```

Ausgabe: `24`

Partielle Auswertung

Gleich wie in Java ist auch hier die Unterstützung des *Currying*s an die Lambda-Funktionen gebunden:

```
Func<int, Func<int, int>> sum = a => (b => a + b);
Func<int, int> sum_ = sum(3);
Console.WriteLine(sum_(3));
```

Typinferenz

Seit C# 3.0 gibt es die Typinferenz auf lokalen Variablen (gleich wie in Java).

```
var x = 10;
```

Der Rest entspricht der Aussage zu Java: **stark und statisch**.

Listen und Tupel

Die Listen und die Collections allgemein ähneln denen von Java und sind im Modul `System.Collections` bzw. `System.Collections.Generic` zu finden.

Was Tupel betrifft, musste man sie vor C# 7.0 gleich wie in Java selbst implementieren. Ab C# 7.0 ist dies nicht länger nötig: https://docs.microsoft.com/en-us/dotnet/csharp/tuples. Ein Beispiel:

```
var left = (a: 5, b: 10);
var right = (a: 5, b: 10);
Console.WriteLine(left == right); // displays 'true'
```

~~Endrekursionsoptimierung~~

Ähnlich der Situation in Java: Gewisse .NET-Sprachen wie F# unterstützen die Endrekursion, C# allgemein aber nicht. Gute Erklärung: https://stackoverflow.com/questions/15864670/generate-tail-call-opcode#15865150

Interessanter Workaround: https://www.thomaslevesque.com/2011/09/02/tail-recursion-in-c/

Musterabgleich

Ab C# 7.0 möglich, so wie es die offizielle Dokumentation von Microsoft bezeugt: https://docs.microsoft.com/en-us/dotnet/csharp/pattern-matching. Ein Beispiel:

```
public static double ComputeArea_Version4(object shape) {
    switch (shape) {
        case Square s when s.Side == 0:
        case Circle c when c.Radius == 0:
        case Triangle t when t.Base == 0 || t.Height == 0:
        case Rectangle r when r.Length == 0 || r.Height == 0:
            return 0;
    ...
    }
}
```

Unveränderliche Variablen

Alle zuvor deklarierten Variablen stehen den Funktionen zur Verfügung. Wenn man spätere Änderungen per `const` ausschließt, erhält man das Verhalten von Java:

```
const int x = 3;
Func<int> f = () => x;
//x = 4; // nicht mehr möglich
Console.WriteLine(f());
```

PHP

Funktionen erster Klasse

Seit PHP 5.3 gibt es die anonymen Funktionen, die überall definiert, übergeben und zurückgeben werden können (http://it2.php.net/manual/de/functions.anonymous.php).

```
$x = function($f) { return $f; };
printf("%s\n", $x(function() { return "Hallo Welt"; })());
```

Allerdings lassen sich auch normale Funktionen überall definieren und übergeben, sodass PHP sie als erstrangig ansieht: http://it2.php.net/manual/de/functions.user-defined.php

```
function x($f) { return $f; }
function retHalloWelt() { return "Hallo Welt"; }
printf("%s\n", x("retHalloWelt")());
```

Neue Syntax ab PHP 7.4:

```
array_map(fn($n) => $n * 10, [1, 2, 3, 4]); // [10, 20, 30, 40]
```

Rekursion

Auch hier ist Rekursion bei normalen Funktionen möglich, bei anonymen Funktionen ist etwas mehr Aufwand nötig:

```
$fak = function($n) { return $n > 1 ? $n * $fak($n-1) : 1; }; //
übersetzt nicht
```

Mit einer `use`-Klausel die eine Referenz auf $fib übergibt sind wir aber erfolgreich:

```
$fak = function($n) use(&$fak) { return $n > 1 ? $n*$fak($n-1) :
1; };
echo $fak(4);
```

Ausgabe: `24`

> Anmerkung: Laut http://it2.php.net/manual/de/functions.user-defined.php und https://www.moreofless.co.uk/tail-call-optimization-elimination-php/ wird nur die normale Rekursion unterstützt, bei der man die **Rekursionstiefe von 100-200 nicht überschreiten** sollte.
> Laut https://stackoverflow.com/questions/4293775/increasing-nesting-function-calls-limit sollte sich dieses Limit mittels einer entsprechenden Direktive erhöhen lassen.

Partielle Auswertung

Auch die Partielle Auswertung ist möglich, dazu muss man im Vorfeld übergebene Argumente in die `use`-Klausel miteinbeziehen:

```
$sum = function($a) { return function($b) use ($a) { return $a+
$b; }; };
printf("%d\n", $sum(3)(3));
```

Typinferenz

PHP verfügt seit jeher über die Typinferenz.

```
$x = 7;
echo gettype($x);
```

Rückgabe: `integer`

```
$x = "Hallo Welt";
echo gettype($x);
```

Rückgabe: `string`

Variablen können ihren Typ problemlos ändern, Konversionen entfallen in vielen Fällen. Diese **schwache und dynamische Typisierung** ist daher bei jener von JS anzusiedeln (http://it2.php.net/manual/de/language.types.intro.php).

Listen und Tupel

Listen können mithilfe von Arrays umgesetzt werden, wie es die Dokumentation von PHP vorgibt: http://it2.php.net/manual/de/language.types.array.php.

Tupel gibt es in PHP nicht als speziellen Datentyp, wofür Arrays oder Objekte herhalten müssen. Ein Beispiel:

```
$x = array( 'name', 'surname', 30 );
printf("%s %s %d", $x[0], $x[1], $x[2]);
```

~~Endrekursionsoptimierung~~

Nicht vorhanden, da schon die Rekursion von Haus aus eingeschränkt ist.

~~Musterabgleich~~

Nur als externes Modul: https://github.com/functional-php/pattern-matching.

Ab PHP 8 existiert das `match`-Statement, welches sich künftig in die richtige Richtung entwickeln könnte: https://www.php.net/manual/en/control-structures.match.php.

Unveränderliche Variablen

PHP unterstützt unveränderliche Variablen mithilfe der `use`-Klausel:

```
$x = 3;
$f = function() use ($x) { return $x; };
$x = 4;
echo $f();
```

Rückgabe: `3`

C++ (und C)

Funktionen erster Klasse

Bis zum Aufkommen der Lambda-Funktionen in C++11 konnte man in C und C++ Funktionen bzw. Methoden nur an genau bestimmten Orten definieren. Es gab lediglich die Funktionszeiger, die es einem erlaubten, eine solche Definition später aufzurufen (Referenz: https://en.wikipedia.org/wiki/Closure_(computer_programming)#Callbacks_(C)).

Ab C++11 (https://en.cppreference.com/w/cpp/language/lambda):

```
auto x = [](auto f){ return f; };
std::cout << x([]{ return "Hallo Welt"; })() << std::endl;
```

Auch eine Kombination mit traditionellen Funktionen ist möglich:

```
const auto retHalloWelt() {
    return "Hallo Welt";
}

int main() {
    auto x = [](auto f){ return f; };
    std::cout << x(retHalloWelt)() << std::endl;
}
```

Rekursion

Auch bei C++ kommt es bei der Rekursion von Lambdas zu Schwierigkeiten, wenn man sie einfach nur so hinschreibt:

```
auto fak = [](int n) { return n > 1 ? n * fak(n-1) : 1; }; // übersetzt nicht
```

Wenn man aber `auto` durch den konkreten Typ (in unserem Fall Closure von int → int) ersetzt und die Referenz von `fak` (`&fak`) in den Kontext übernimmt, dann klappt es (https://stackoverflow.com/questions/2067988/recursive-lambda-functions-in-c11):

```
std::function<int(int)> fak = [&fak](int n) { return n > 1 ? n * fak(n-1) : 1; };
std::cout << fak(4) << std::endl;
```

Ausgabe: `24`

Partielle Auswertung

Für das Currying ist die C++11 Lambda-Syntax Voraussetzung. Es ist wichtig, die einzelnen Argumente den verschachtelten Funktionen im Kontext mitzugeben (`[]`, ähnlich dem PHP-Schlüsselwort `use`):

```
auto sum = [](auto a) { return [a](auto b) { return a + b; }; };
std::cout << sum(3)(3) << std::endl;
```

Typinferenz

Ab C++11 ist die Typinferenz mittels des Schlüsselwortes `auto` an vielen Stellen möglich (lokale Variablen, Parameter, Rückgabewerte), jedoch nicht in Typdefinitionen (Klassen, Strukturen): https://en.cppreference.com/w/cpp/language/auto. Beispiel:

```
auto x = 3;
```

Der Rest entspricht der Aussage zu Java: **stark und statisch**.

C verfügt über keinerlei Typinferenz und Referenztypen gelten als schwach, da leicht umtauschbar (man spricht auch von „Zeiger-Etiketten“). C++ hingegen setzt von `void*` zu `MeinTyp*` stets *einen expliziten Cast* voraus.

Listen und Tupel

C++ verfügt über vielfältige vordefinierte Listentypen, allerdings mit anderer Schnittstelle: https://en.cppreference.com/w/cpp/container.

Ab Version 11 existieren vordefinierte Paar- und Tupeltypen wie `pair` und `tuple` (https://en.cppreference.com/w/cpp/utility/tuple). Beispiel:

```
auto x = std::make_tuple("name", "surname", 30);
std::cout << std::get<0>(x) << " " << std::get<1>(x) << " " << std::get<2>(x) << std::endl;
```

C bietet standardmäßig praktisch nichts, es muss alles händisch definiert werden: Listen über Zeiger, Tupel über Strukturen oder Felder.

Endrekursionsoptimierung

Die optionale Endrekursionsoptimierung muss bei C als auch C++ vom Compiler zur Verfügung gestellt werden. Bei gcc/g++, LLVM/clang, MS VC++ und Intel kommt sie bei der Angabe einer entsprechenden Optimierungsoption zum Tragen (bspw. `-O2`).

Quellen: https://stackoverflow.com/questions/2693683/tail-recursion-in-c, https://en.wikipedia.org/wiki/Tail_call, https://stackoverflow.com/questions/34125/which-if-any-c-compilers-do-tail-recursion-optimization.

Musterabgleich

Derzeit nur über externe Lösungen möglich: https://www.codeproject.com/Articles/1096251/Cplusplus-pattern-matching-type-matching oder auch https://github.com/mpark/patterns, eine künftige Unterstützung ist jedoch geplant: http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2019/p0592r4.html.

Unveränderliche Variablen

Unveränderliche Variablen sind ähnlich wie PHP umsetzbar, man benutze Lambdas mit einer entsprechenden Kontextbindung:

```
auto x = 3;
auto f = [x] { return x; };
x = 4;
std::cout << f() << std::endl;
```

Rückgabe: `3`

Zusammenfassung

- **Python und C++ Gewinner** unter den analysierten Sprachen
  - Bei C++ gibt es Pattern Matching über umfassende externe Bibliotheken, später vielleicht direkt on-board?
  - Bei Python wird die Endrekursion nur bei Vorhandensein einer externen Bibliothek optimiert, für erweiterten Musterabgleich braucht es ebenfalls ein Extra
  - *Anmerkung zu C++*: Aus Performanzgründen hat sich C++ nie dafür entschieden, einen Garbage Collector als eigenen Thread zu integrieren. Die Umsetzung einer automatischen Speicherverwaltung kann auf Wunsch des Programmierers mittels „Out-of-Scoping“ oder der Smart Pointer erfolgen. Dabei kommen jeweils Destruktoren zum Einsatz.
- **Es folgen: JavaScript/ECMAScript, Java und C#**
  - Je ein vollständig fehlendes und zwei teilweise umgesetzte Konzepte
- **Schwächer: PHP**
  - Zwei vollständig fehlende und zwei teilweise implementierte Konzepte
  - Die Rekursionslimits sollten erhöht und die Endrekursionsoptimierung eingeführt werden
- **Das absolute Schlusslicht ist C**, welches nur Rekursion inkl. Endrekursionsoptimierung und Funktionszeiger kennt, nicht mal automatische Speicherverwaltung und Ausnahmen (hier unter *Vorausgesetzte Kriterien*)