

Erlang

so trocken wie Knäckebrot.

Moritz Haarmann

- 1 Übersicht
- 2 Die Sprache
 - Konzepte
 - Datentypen
- 3 Funktionen
- 4 Verteilt und Nebenläufig
- 5 ...

Was ist Erlang?

- ErLang steht für Ericsson Language

Was ist Erlang?

- ErLang steht für Ericsson Language
- Eine funktionale Programmiersprache, entwickelt für Kommunikationssysteme

Was ist Erlang?

- ErLang steht für Ericsson Language
- Eine funktionale Programmiersprache, entwickelt für Kommunikationssysteme
- Bytecode (vgl. Java), dadurch portabel

Was ist Erlang?

- ErLang steht für Ericsson Language
- Eine funktionale Programmiersprache, entwickelt für Kommunikationssysteme
- Bytecode (vgl. Java), dadurch portabel
- Wurde 1986 u.A. von Joe Armstrong in den Ericsson Labs entwickelt

Was ist Erlang?

- ErLang steht für Ericsson Language
- Eine funktionale Programmiersprache, entwickelt für Kommunikationssysteme
- Bytecode (vgl. Java), dadurch portabel
- Wurde 1986 u.A. von Joe Armstrong in den Ericsson Labs entwickelt
- Die Sprache und die Open Telecommunications Platform zusammen bilden die Laufzeitumgebung.

Quicksort in Erlang

```
1 -module(qsort).
2 -export([qsort/1]).
3
4 qsort([]) -> [];
5 qsort([Pivot|T]) -> qsort([X || X<-T, X<Pivot])
6                   ++ [Pivot] ++
7                   qsort([X || X<-T, X>=Pivot]).
```

Eine funktionale Sprache

- Aufbau allein durch Komposition von Funktionen

Eine funktionale Sprache

- Aufbau allein durch Komposition von Funktionen
- Anonyme Funktionen, Funktionen höherer Ordnung, Funktionen als Parameter

Eine funktionale Sprache

- Aufbau allein durch Komposition von Funktionen
- Anonyme Funktionen, Funktionen höherer Ordnung, Funktionen als Parameter
- Vermeidung von Seiteneffekten!

Eine funktionale Sprache

- Aufbau allein durch Komposition von Funktionen
- Anonyme Funktionen, Funktionen höherer Ordnung, Funktionen als Parameter
- Vermeidung von Seiteneffekten!
- Alles hat einen Wert, kein explizites Return-Statement

Eine funktionale Sprache

- Aufbau allein durch Komposition von Funktionen
- Anonyme Funktionen, Funktionen höherer Ordnung, Funktionen als Parameter
- Vermeidung von Seiteneffekten!
- Alles hat einen Wert, kein explizites Return-Statement
- Single Assignments: Variablen werden einmal mit einem Wert belegt, der danach nicht mehr geändert werden kann (vgl. Java's final-Keyword)

Eine nebenläufige Sprache

- Prozesse sind sehr ressourcenschonend implementiert

Eine nebenläufige Sprache

- Prozesse sind sehr ressourcenschonend implementiert
- Sämtliche Kommunikation der Prozesse untereinander erfolgt durch *Message Passing*

Eine nebenläufige Sprache

- Prozesse sind sehr ressourcenschonend implementiert
- Sämtliche Kommunikation der Prozesse untereinander erfolgt durch *Message Passing*
- Durch *Message Passing* und das Wegfallen von Seiteneffekten sind keine Locks oder andere Sperrmechanismen notwendig, um Konflikte zu vermeiden.

Eine verteilte Sprache

- Einfache Kommunikation verschiedener *Knoten*

Eine verteilte Sprache

- Einfache Kommunikation verschiedener *Knoten*
- Die OTP stellt verschiedene Mechanismen bereit, um die Entwicklung verteilter Anwendungen sehr zu beschleunigen, z.B. allgemeine Server.

Eine verteilte Sprache

- Einfache Kommunikation verschiedener *Knoten*
- Die OTP stellt verschiedene Mechanismen bereit, um die Entwicklung verteilter Anwendungen sehr zu beschleunigen, z.B. allgemeine Server.
- Knoten können sich gegenseitig überwachen, oder aber auf Einrichtungen der OTP zurückgreifen, die beispielsweise vollständige Überwachungs bäume bereitstellen.

Eine verteilte Sprache

- Einfache Kommunikation verschiedener *Knoten*
- Die OTP stellt verschiedene Mechanismen bereit, um die Entwicklung verteilter Anwendungen sehr zu beschleunigen, z.B. allgemeine Server.
- Knoten können sich gegenseitig überwachen, oder aber auf Einrichtungen der OTP zurückgreifen, die beispielsweise vollständige Überwachungs bäume bereitstellen.
- Auch hier werden Daten über Message Passing verschickt, Byte Order Conversion oder andere typische Probleme werden von Erlang gemeistert.

Pattern Matching

- Die Zuweisung von Werten anhand ihrer Struktur wird *Pattern Matching* genannt.

Pattern Matching

- Die Zuweisung von Werten anhand ihrer Struktur wird *Pattern Matching* genannt.
- Kernelement von Erlang, wird z.B. bei Funktionsaufrufen, Wertzuweisungen, Nachrichtenabwicklung genutzt.

Pattern Matching

- Die Zuweisung von Werten anhand ihrer Struktur wird *Pattern Matching* genannt.
- Kernelement von Erlang, wird z.B. bei Funktionsaufrufen, Wertzuweisungen, Nachrichtenabwicklung genutzt.
- explizit

```
1> z = 1.  
1
```

Pattern Matching

- Die Zuweisung von Werten anhand ihrer Struktur wird *Pattern Matching* genannt.
- Kernelement von Erlang, wird z.B. bei Funktionsaufrufen, Wertzuweisungen, Nachrichtenabwicklung genutzt.
- explizit

```
1> z = 1.  
1
```

Pattern Matching

- Die Zuweisung von Werten anhand ihrer Struktur wird *Pattern Matching* genannt.
- Kernelement von Erlang, wird z.B. bei Funktionsaufrufen, Wertzuweisungen, Nachrichtenabwicklung genutzt.
- explizit

```
1> Z = 1.  
1
```

- implizit

```
flaeche({rechteck, X, Y}) -> X*Y,  
flaeche({quadrat, X}) -> X*X.
```

Atom

- Ein *Atom* ist ein nichtzerlegbarer Ausdruck, wie z.B.

```
atom javaistdiebestesprachederwelt erlang 'rocknroll'.
```

Atom

- Ein *Atom* ist ein nichtzerlegbarer Ausdruck, wie z.B.

```
atom javaistdiebestesprachederwelt erlang 'rocknroll'.
```

- Nützlich z.B. für implizites Pattern Matching

Atom

- Ein *Atom* ist ein nichtzerlegbarer Ausdruck, wie z.B.

```
atom javaistdiebestesprachederwelt erlang 'rocknroll'.
```

- Nützlich z.B. für implizites Pattern Matching
- Ein Atom hat sich selbst als Wert, d.h. rocknroll hat den Wert rocknroll

Zahlen

- Ganzzahlen: Wie gewohnt, jedoch gibt es auch die Möglichkeit, die Basis mit einzubeziehen:

```
1> 16#affe + 32#tiger + $c + 12.  
31060552  
2> $c.  
99
```

ist gültiger Code!

Zahlen

- Ganzzahlen: Wie gewohnt, jedoch gibt es auch die Möglichkeit, die Basis mit einzubeziehen:

```
1> 16#affe + 32#tiger + $c + 12.  
31060552  
2> $c.  
99
```

ist gültiger Code!

- Ganzzahlen haben keine feste Größe. Die tatsächlich verfügbare Größe wird nur von der Laufzeitumgebung begrenzt.

Zahlen

- Ganzzahlen: Wie gewohnt, jedoch gibt es auch die Möglichkeit, die Basis mit einzubeziehen:

```
1> 16#affe + 32#tiger + $c + 12.  
31060552  
2> $c.  
99
```

ist gültiger Code!

- Ganzzahlen haben keine feste Größe. Die tatsächlich verfügbare Größe wird nur von der Laufzeitumgebung begrenzt.
- Gleitkommazahlen werden mit 64 Bit gespeichert, IEEE754-konform (Java: Double)

Tupel

- Eine Tupel ist ein anonymer Elementcontainer mit fester Größe

```
1> X = {rechteck,12,10}.  
{rechteck,12,10}  
2> {_,Breite,Hoehe} = X.
```

Tupel

- Eine Tupel ist ein anonymer Elementcontainer mit fester Größe

```
1> X = {rechteck,12,10}.  
{rechteck,12,10}  
2> {_,Breite,Hoehe} = X.
```

- Einzelne Elemente können einfach extrahiert und gematcht werden.

Tupel

- Eine Tupel ist ein anonymer Elementcontainer mit fester Größe

```
1> X = {rechteck,12,10}.  
{rechteck,12,10}  
2> {_,Breite,Hoehe} = X.
```

- Einzelne Elemente können einfach extrahiert und gematcht werden.
- Nur lesbar, d.h. eine vorhandene Tupel kann nicht verändert werden.

Tupel

- Eine Tupel ist ein anonymer Elementcontainer mit fester Größe

```
1> X = {rechteck,12,10}.  
{rechteck,12,10}  
2> {_,Breite,Hoehe} = X.
```

- Einzelne Elemente können einfach extrahiert und gematcht werden.
- Nur lesbar, d.h. eine vorhandene Tupel kann nicht verändert werden.
- Erweiterung sind Records, Tupeln mit benannten Elementen.

Listen

- Listen sind sehr wichtig, deshalb gute Integration

```
1> X = [1,2,3,4,5,6,7,8,9,10].  
[1,2,3,4,5,6,7,8,9,10]  
2> L = [ Z || Z<-X,Z>5 ].  
[6,7,8,9,10]
```

Listen

- Listen sind sehr wichtig, deshalb gute Integration

```
1> X = [1,2,3,4,5,6,7,8,9,10].  
[1,2,3,4,5,6,7,8,9,10]  
2> L = [ Z || Z<-X,Z>5 ].  
[6,7,8,9,10]
```

- Listen können einfach verändert werden, in dem Elemente angehängt oder entfernt werden

Listen

- Listen sind sehr wichtig, deshalb gute Integration

```
1> X = [1,2,3,4,5,6,7,8,9,10].  
[1,2,3,4,5,6,7,8,9,10]  
2> L = [ Z || Z<-X,Z>5 ].  
[6,7,8,9,10]
```

- Listen können einfach verändert werden, indem Elemente angehängt oder entfernt werden
- Eine Liste besteht immer aus dem Kopf und einem Schwanz

```
3> [U|V] = L.  
[6,7,8,9,10]  
4> U.  
6
```

Syntax

- Funktionen bestehen aus einem Kopf und einem Körper

```
sum(X, Y) -> X+Y.
```

Syntax

- Funktionen bestehen aus einem Kopf und einem Körper

```
sum(X, Y) -> X+Y.
```

- Sie können mehrere Sätze beinhalten

```
pop([]) -> [],  
pop([H|_]) -> H.  
% allersinnlosestes codebeispiel.
```

Syntax

- Funktionen bestehen aus einem Kopf und einem Körper

```
sum(X, Y) -> X+Y.
```

- Sie können mehrere Sätze beinhalten

```
pop([]) -> [],  
pop([H|_]) -> H.  
% allersinnlosestes codebeispiel.
```

- Passender Satz wird über Pattern Matching gefunden

Spezielles

- Guards: Schränken das Pattern Matching im Funktionskopf weiter ein

```
max(X, Y) when X>Y -> X,  
max(X, Y) -> Y.
```

Spezielles

- Guards: Schränken das Pattern Matching im Funktionskopf weiter ein

```
max(X,Y) when X>Y -> X,  
max(X,Y) -> Y.
```

- Tail-Recursion: Eine Rekursion am Ende einer Funktion müllt den Stack nicht zu.

```
taily() ->  
io:format("nicht schlimm~n"),  
taily().
```

Spezielles

- Guards: Schränken das Pattern Matching im Funktionskopf weiter ein

```
max(X,Y) when X>Y -> X,  
max(X,Y) -> Y.
```

- Tail-Recursion: Eine Rekursion am Ende einer Funktion müllt den Stack nicht zu.

```
taily() ->  
io:format(" nicht schlimm~n"),  
taily().
```

- Grundsätzlich wird eine Exception geworfen, wenn keine Funktion passt

Quicksort in Erlang

```
1 -module(qsort).  
2 -export([qsort/1]).  
3  
4 qsort([]) -> [];  
5 qsort([Pivot|T]) -> qsort([X || X<-T, X<Pivot])  
6                     ++ [Pivot] ++  
7                     qsort([X || X<-T, X>=Pivot]).
```

Einfache Verteilung

- `spawn(Funktion)` startet einen Prozess, der die genannte Funktion ausführt, und gibt eine Prozesskennung (`Pid`) zurück.

Einfache Verteilung

- `spawn(Funktion)` startet einen Prozess, der die genannte Funktion ausführt, und gibt eine Prozesskennung (`Pid`) zurück.
- der Operator `!` sendet eine Nachricht an einen Prozess

Einfache Verteilung

- `spawn(Funktion)` startet einen Prozess, der die genannte Funktion ausführt, und gibt eine Prozesskennung (`Pid`) zurück.
- der Operator `!` sendet eine Nachricht an einen Prozess
- `receive ... end.` Empfängt Nachrichten

Einfache Verteilung

```
-module(simple).  
-export([loop/0]).  
  
loop()->  
  receive  
    hallo ->  
      io:format("guten tag!"),  
      loop();  
    tschuess ->  
      io:format(" auf wiedersehen!"),  
      loop()  
  
end.
```

```
1> Z = spawn(fun() -> simple:loop() end).  
   <0.33.0>  
2> Z ! hallo.  
   guten tag!hallo
```

Mehr cooles Zeug!

- Bit-Syntax, einfache Syntax um binäre Daten zu formatieren und einzelne Bits zu “matchen”

```
4> Z = <<X:1,Y:6,Z:1>>.
<<156>>
```

Mehr cooles Zeug!

- Bit-Syntax, einfache Syntax um binäre Daten zu formatieren und einzelne Bits zu “matchen”

```
4> Z = <<X:1,Y:6,Z:1>>.
<<156>>
```

- Hot Swapping: Code kann zur Laufzeit geändert werden, ohne dass eine Unterbrechung der Ausführung nötig ist.

Mehr cooles Zeug!

- Bit-Syntax, einfache Syntax um binäre Daten zu formatieren und einzelne Bits zu “matchen”

```
4> Z = <<X:1,Y:6,Z:1>>.
<<156>>
```

- Hot Swapping: Code kann zur Laufzeit geändert werden, ohne dass eine Unterbrechung der Ausführung nötig ist.
- ...so viel mehr!

Zum Weiterlesen

- www.erlang.se, offizielle Seite

Zum Weiterlesen

- www.erlang.se, offizielle Seite
- armstrongsoftware.blogspot.com, Blog von Joe Armstrong

Zum Weiterlesen

- www.erlang.se, offizielle Seite
- armstrongsoftware.blogspot.com, Blog von Joe Armstrong
- und natürlich: [Programming Erlang](#) von Joe Armstrong

Fragen?

Danke!