>Combinator Parsing in Scala

Mit Scala Parser für die JVM einfach entwickeln

Arbeitskreis Objekttechnologie Norddeutschland
HAW Hamburg, 21. Juni 2010

Stephan Freund

.comdirect

> Agenda

- 1. Scala im Überblick
- 2. Parser Generator oder Combinator Parser?
- 3. Einfacher Parser für JSON
- 4. Implementierung von Combinator Parsers
- 5. DSL und Parser für UML Statecharts
- 6. Abschluss

> Vorstellung

- 1973 in Calgary, Alberta, Kanada geboren
- Studium der theoretischen Informatik an der **Queen's University** in Kingston, Ontario
- 12 Jahre Erfahrung mit Java als Entwickler und Berater
- Seit 2002 in Deutschland berufstätig
- Bei **comdirect Bank AG** in Quickborn, Gruppe *Frameworks & Tools*

> 1. Scala im Überblick

> Moderne Sprache für die JVM

- Mit Java voll kompatibel
  - Kompiliert zu Java Bytecode
  - Java Bibliotheken und Klassen können genutzt werden
- Läuft auf JVM
  - Scala Runtime Bibliothek neben Java Klassen auf der Java VM
  - Vorteile von Sun HotSpot JIT-Kompilierung bleiben
  - Wie andere JVM-Sprachen: JRuby, Clojure, Groovy, Jython ...

> Multi-Paradigm

- Objekt-Orientiert
  - Primitive Typen, Arrays sind auch Klassen
  - Universelle Klassenhierarchie
- Funktional
  - Funktionen sind auch Objekte
  - Higher-Order Funktionen, Currying, anonyme Funktionen, Closures alle mit einer kompakten Syntax möglich
- Statische Typisierung
  - Mächtige Typ-Inferenz erlaubt an vielen Stellen implizite Typisierung
  - Sowohl Typ-Parametrisierung (wie Java Generics) und abstrakte Typ-Felder

> Skalierbar

„Für kleine und große Anwendungen“

- Kleine Anwendungen und Scripting
  - *REPL* (Read-Evaluate-Print-Loop) Interpreter für die Konsole
  - Knappe Syntax, für kleine Skripte und Textbearbeitung geeignet
- Skalierbar für große Anwendungen durch Erweiterbarkeit
  - Eigene Kontrollstrukturen definieren
  - Interne DSL werden stark unterstützt

> Eine funktionale Sprache

- Alle Funktionen sind Objekte
  - Anonyme Funktionen, Currying und Nested Funktionen sind möglich
- Scala bietet Higher-Order Funktionen
  - Eine Funktion kann Funktionen als Parameter oder Return-Typ haben

```
def exists[T](xs: Array[T], p: T => Boolean) = {
    var i: Int = 0
    while (i < xs.length && !p(xs(i))) i = i + 1
    i < xs.length
}
def forall[T](xs: Array[T], p: T => Boolean) = {
    def not_p(x: T) = !p(x)
    !exists(xs, not_p)
}
```

> Funktion-Typen sind spezielle Klassen

- Funktionen sind Instanzen von Klassen mit `apply` Methoden
- Der Funktion Type `S => T` entspricht dem Class Type `Function1`:

```
package scala
abstract class Function1[-S, +T] {
   def apply(x: S): T
}
```

- Äquivalente anonyme Funktionen:

```
(x: Int) => x + 1

new Function1[Int, Int] {
   def apply(x: Int): Int = x + 1
}
```

>Unterstützung für DSLs

„Syntactic Sugar“ für die Entwicklung von internen Domain Spezifische Sprachen (DSLs) oder Fluent Interfaces

- In der Syntax optionale Elemente: Semikolon, Punkt, Klammern
- Implizite Parameter
- Symbolische Methodennamen
- Funktion Objekte

```
def isEven(n: Int) = (n % 2) == 0
List(1, 2, 3, 4) filter isEven foreach println
```

statt

```
(new List(1,2,3,4)).filter(isEven(_)).foreach(println(_))
```

> 2. Parser Generator oder Combinator?

> Parser-Arten und externe DSL

- Parser Generatoren
  - yacc/bison generiert einen LALR Parser in C
  - ANTLR generiert einen LL(*) Parser in Java
  - sehr schnell z.B. durch Lookup-Tabellen
- PEG (Parsing Expression Grammar) und Packrat Parser
  - PetitParser für Smalltalk
  - Schnell durch Memoization, aber Speicher-Intensiv
- Einsatzgebiet externe DSL
  - Auf einen Fachgebiet zugeschnitten
  - Externe Input-Dateien werden gelesen

> Combinator Parser

- In funtionalen Sprachen wie Haskell als funktionales Parsing bekannt
- Ein Baukasten für Parser
  - Parser für spezifische Inputarten, z.B. Gleitkommazahlen oder Ganzzahlen, können kombiniert werden, um andere Kombinatoren für größere Ausdrücke zu bilden
- Parser-Typ ist LL (top-down) mit Backtracking
  - Left-Recursive Grammatik nicht möglich
  - Backtracking ist relativ langsam
- Lesbar, Modular, Wartbar
  - Im Programm eingebettet
  - Kein Generator-Schritt
  - Alle Sprachmittel können genutzt werden

> 3. Einfacher Parser für JSON

> Beispiel JSON

- JavaScript Object Notation – Datentransferformat für AJAX Webseiten
- JSON Daten: ein Adressbuch

```
{
  "address book": {
    "name": "John Smith",
    "address": {
      "street": "10 Market Street",
      "city"  : "San Francisco, CA",
      "zip"   : 94111
    },
    "phone numbers": [
      "408 338-4238",
      "408 111-6892"
    ]
  }
}
```

>JSON Grammatik

- eBNF-Syntax
- `[ … ]` ist eine Liste
- `{ … }` ist optionale Wiederholung

```
value   = obj | arr | stringLiteral | floatingPointNumber |
            "null" | "true" | "false";
obj     = "{", [ members ], "}";
arr     = "[", [ values ], "]";
members = member, {",", member};
member  = stringLiteral, ":", value;
values  = value, {",", value};
```

>Simple JSON Parser

```
import scala.util.parsing.combinator._

class JSON extends JavaTokenParsers {

  def value : Parser[Any] = obj | arr |
                            stringLiteral |
                            floatingPointNumber |
                            "null" | "true" | "false"

  def obj   : Parser[Any] = "{"~repsep(member, ","~)"}"

  def arr   : Parser[Any] = "["~repsep(value, ","~)"]"

  def member: Parser[Any] = stringLiteral~":"~value
}
```

>Simple JSON Parser – Ausführung und Ergebnis

```
import java.io.FileReader

object ParseJSON extends JSON {
  def main(args: Array[String]) {
    val reader = new FileReader(args(0))
    println(parseAll(value, reader))
  }
}
```

```
$ scala ParseJSON address-book.json

[13.4] parsed: (({~List((("address book"~:)~(({~List(((
"name"~:)~"John Smith"), (("address"~:)~(({~List(((
"street"~:)~"10 Market Street"), (("city"~:)~"San Francisco
,CA"), (("zip"~:)~94111)))~})), (("phone numbers"~:)~(([~
List("408 338-4238", "408 111-6892"))~])))))~})))))~})
```

>JSON Parser mit Konversion in Map

```
import scala.util.parsing.combinator._

class JSON1 extends JavaTokenParsers {

  def obj: Parser[Map[String, Any]] =
    "{"~> repsep(member, ",") <~"}" ^^ (Map() ++ _)

  def arr: Parser[List[Any]] = "["~> repsep(value, ",") <~"]"

  def member: Parser[(String, Any)] =
    stringLiteral~":"~value ^^
      { case name~":"~value => (name, value) }

  def value: Parser[Any] = (obj | arr | stringLiteral
  | floatingPointNumber ^^ (_.toDouble)
  | "null"  ^^ (x => null)
  | "true"  ^^ (x => true)
  | "false" ^^ (x => false)
  )
}
```

>JSON Parser mit Konversion – Ergebnis

```
$ scala JSON1Test address-book.json

[14.1] parsed: Map(
  address book -> Map(
    name -> John Smith,
    address -> Map(
      street -> 10 Market Street,
      city -> San Francisco, CA,
      zip -> 94111),
    phone numbers -> List(408 338-4238, 408 111-6892)
  )
)
```

>Parser Kombinatoren

`"..."`	literal
`"...".r`	regular expression
`P~Q`	sequential composition
`P <~ Q, P ~> Q`	sequential composition; keep left/right only
`P | Q`	alternative
`opt(P)`	option
`rep(P)`	repetition
`repsep(P, Q)`	interleaved repetition
`P ^^ f`	result conversion

>Regular Expression Parsers

- Syntax Analyse hat normalerweise zwei Phasen: Lexer und Parser
  - Token-Erkennung und syntaktische Analyse
- In Scala sind Regular Expressions im Parser oft ausreichend
- `JavaTokenParsers` hat Parser für Standard-Tokens:

```
def identifier: Parser[String] = """[a-zA-Z_]\w*""".r

def wholeNumber: Parser[String] = """-?\d+""".r

def decimalNumber: Parser[String] = """(\d+(\.\d*)?|\d*.\d+)""".r
```

> 4. Implementierung von Combinator Parsers

> Parser sind kombinierbare Funktionen

- Combinators sind Parser-Funktionen, die miteinander kombiniert und zu größeren Parsern kombiniert werden können

- Ein Parser ist eine Funktion,
  - die einen Teil des Inputs konsumiert und den Rest zurückliefert
  - von einem Input-Typ zu einem Result-Typ
  - die einer Grammatik entspricht und Erfolg oder Fehler liefert

```
type Parser[T] = Input => ParseResult[T]
abstract class ParseResult[+T]
case class Success[T] (result: T, in: Input) extends
   ParseResult[T]
case class Failure(msg: String, in: Input) extends
   ParseResult[Nothing]
```

> Character Parser

- Der Typ-Parameter `Elem` ist der Token-Type, der Ergebnis-Typ des Parsers

```
type Elem = Char
type Input = Reader[Char]

def elem(kind: String, p: Elem => Boolean) =
   new Parser[Elem] {
      def apply(in: Input) =
         if (p(in.first)) Success(in.first, in.rest)
         else Failure("expected " + kind, in)
   }

def letter = elem("letter", _.isLetter)
def digit = elem("digit", _.isDigit)
```

> Die Parser-Oberklasse

- Weil ein Parser eine Funktion ist (d.h. erbt von), muss er eine `apply` Methode definieren
- `Input => ParseResult[T]` ist eine Abkürzung für `scala.Function1[Input, ParseResult[T]]`

```
abstract class Parser[+T] extends (Input => ParseResult[T])
{  p =>  // synonym for this
   def apply(in: Input): ParseResult[T]
   def ~  …   // concatenation
   def |  …   // alternation
   def ^^ …   // result transformation
   …
}
```

> Sequential Composition

- `P~Q` ist ein Parser, der zuerst P versucht. Wenn P erfolgreich ist, wird Q versucht.
- `p =>` is ein Alias für `this`, d.h. hier P, der linke Operand oder Receiver von `~`
- `new ~(x, y)` ist eine Instanz der Case-Klasse mit dem Namen `~`
- Alternation `P|Q` ist ähnlich: wenn P nicht erfolgreich ist, wird Q versucht

```
def ~ [U](q: => Parser[U]) = new Parser[T~U] {
   def apply(in: Input) = p(in) match {
      case Success(x, in1) =>
         q(in1) match {
            case Success(y, in2) => Success(new ~(x, y), in2)
            case failure => failure
         }
      case failure => failure
   }
}
```

> Result Conversion

- Der Kombinator `^^` konvertiert das Parse-Ergebnis z.B. in Scala Objekte zur Weiterbearbeitung
- Einfach implementiert: `P ^^ f` ist erfolgreich genau dann, wenn P erfolgreich ist

```
def ^^ [U](f: T => U): Parser[U] = new Parser[U] {
   def apply(in: Input) = p(in) match {
      case Success(x, in1) => Success(f(x), in1)
      case failure => failure
   }
}
```

> 5. UML Statechart Parser

> DSL für UML Statecharts

- Ohne Zustandsmaschinen wäre das Beschreiben des Verhaltens von einem komplexen System unbeherrschbar
- UML Statechart erweitert die übliche Zustandsmaschine um verschachtelte Zustände und orthogonale Regionen
  - um Zustands- und Transitions-Explosion zu verhindern
  - Von Harel Statecharts
- Einsatzgebiete vielfach, z.B. Gerätesteuerung, Orderausführung
- Eine Externe DSL wird implementiert mit einer Microwelle als Beispiel

> Statechart Artefakte

- Grammatik in eBNF Form: `statechart-grammar.txt`
- Parser ohne Konversionen: `SimpleStatechartParser.scala`
- Parser mit Konversionen: `StatechartParser.scala`
- Statechart Implementierung: `Statechart.scala`
- Beispiel Mikrowelle – Diagramm: `microwave-statechart.png`
- Beispiel Mikrowelle – Statechart Textbeschreibung: `microwave.txt`

> Statechart Klassen

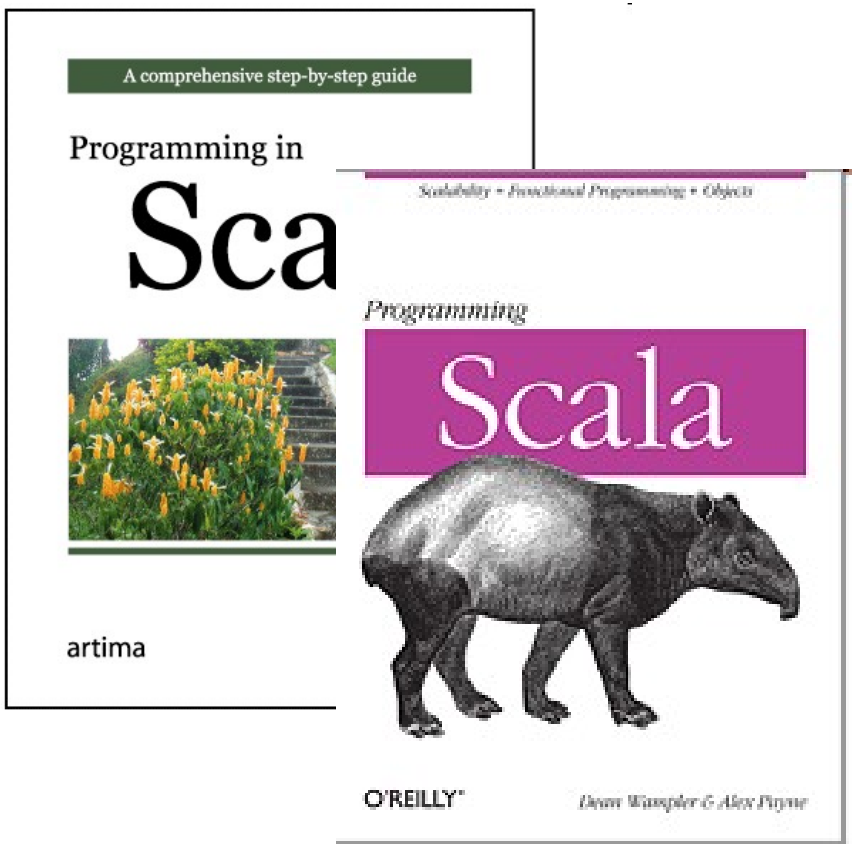
```
class Statechart(name: String, properties: Map[String,Any],
    initialState: String, states: List[State]) {
  var currentState
  def start()
  def trigger(event: String)
}
class State(name: String, onEntry: Any, onExit: Any,
    outgoingTransitions: List[Transition])

class ExtendedState(name: String, substates: List[String],
    initialState: String, outgoingTransitions: List[Transition]) {
  var currentState
}
class Transition(event: String, guard: Option[Condition],
    nextState: String)

class Condition(not: Option[_], ident: String, comp: Comparison)
```

> 6. Abschluss

>Fazit

- Übergangslose Integration
  - Im Programm eingebettet
  - Eine „ausführbare Spezifikation“
- Ineffizient im Vergleich mit Parser Generatoren
  - Parser Konstruktion und Input-Analyse sind vermischt
  - Viele Parser-Objekte werden für jede Analyse erzeugt
  - Parser Generatoren generieren schnelle Lookup-Tabellen
- Keine zusätzliche Syntax, sondern eine Bibliothek (eine interne DSL)
- Scala lohnt sich!
  - Bietet viele Möglichkeiten, die man sonst in Java nicht hat
  - Eine Weiterentwicklung von Java, durchdacht und einheitlich
  - Anspruchsvoll: Typ-Inferenz und Flexibilität des Ausdrucks

> Sourcen und Ressourcen



Martin Odersky

- Erfinder von Scala
- Schrieb Java 1.4 Compiler, Generics
- Seit 1999 Professor an der EPFL

Scala Home

- `http://www.scala-lang.org/`

Language Research

- `http://www.scala-lang.org/node/143`

Parser Combinators

Moors, Piessens und Odersky

- `http://www.cs.kuleuven.be/publicaties/rapporten/cw/CW491.pdf`

`http://booksites.artima.com/programming_in_scala/`
`http://programming-scala.labs.oreilly.com/`