Compilerbau CS1019

Th. Letschert

TH Mittelhessen Gießen

University of Applied Sciences

Syntaxanalyse mit Parserkombinatoren

– Die Idee der Parser-Kombinatoren

– Parserkombinatoren in Scala

Die Idee der Parser-Kombinatoren

Parser-Kombinatoren

- Parser-Kombinatoren kombinieren Parser zu neuen Parsern.
- Ein Parser-Kombinator nimmt Parser und macht daraus einen neuen Parser
  (Der Kombinator kombiniert Parser zu einem neuen Parser)

Idee der Parserkombinatoren

Parser werden aus anderen Parsern mit Hilfe von „Kombinatoren“ erzeugt,

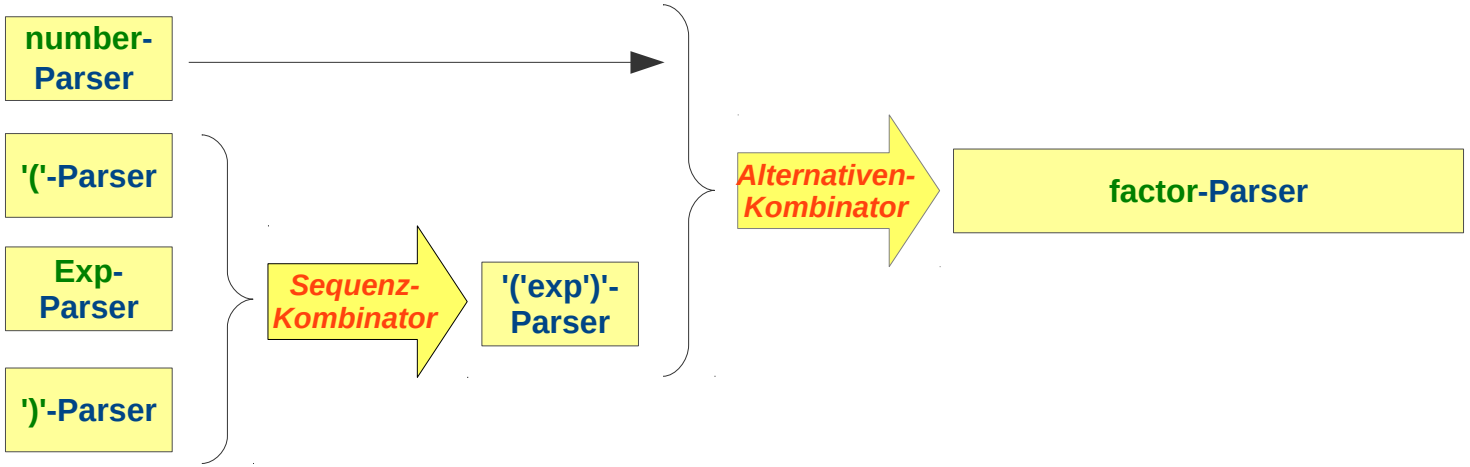
Ein Parser für ein komplexes Konstrukt wird mit

- einem (Parser-) Kombinator
- aus Parsern für einfachere Konstrukte

erzeugt.

Beispiel

factor ::= *number* | '(' *exp* ')'



Bestandteile einer kontextfreien Grammatik

Produktionen

Eine Grammatik enthält Produktionen der Form:

$$A \rightarrow B_1 \; B_2 \ldots B_n$$

A ist dabei ein Nonterminal die B_i sind Terminale oder Nonterminale

Alternativen

Produktionen mit der gleichen linken Seite werden zu Alternativen zusammengefasst:

$$A \rightarrow P_1 \mid P_2 \mid \ldots \mid P_n$$

A ist dabei ein Nonterminal und die P_i sind rechte Seiten von Produktionen

Konstruktion eines Parsers

Grammatik

Enthält Regeln der Form

$$A \rightarrow B_1 \; B_2 \ldots B_n$$

$$A \rightarrow P_1 \mid P_2 \mid \ldots \mid P_n$$

Parser für eine Grammatik

Ein Parser für eine solche Grammatik kann man konstruieren

- aus jeweils einen Parser für jedes Terminal
- mit diesen kann man einen Parser für jede rechte Seite (Body) einer Produktion

  $B_1 \; B_2 \ldots B_n$

  konstruieren
- und aus diesen einen Parser für die Alternativen jedes Nonterminals

  $A \rightarrow P_1 \mid P_2 \mid \ldots \mid P_n$
- Der Parser für das Startsymbol ist dann der gesuchte Parser

Konstruktion eines Parsers aus einer Grammatik

Parser für Terminale

Parser für Terminale können mit den üblichen Verfahren für die Erkennung regulärer Sprachen realisiert werden.

A → [1-9][0-9]*

Parser für Produktionen

Parser für Produktionen müssen eine Folge / Sequenz von Terminalen und Nichtterminalen erkennen.

Hat man einen Parser für jeden Bestandteil einer Produktion, dann kann man aus ihnen Parser für die gesamte Produktion konstruieren.

A → XY

Parser für Nonterminale / Alternativen

Parser für Nichtterminale müssen eine von mehreren alternativen Produktionen erkennen.

Hat man für jede Produktion einen Parser für jede Alternative, dann kann man aus ihnen einen Parser das Nichtterminal konstruieren.

A → X | Y

Konstruktion eines Parsers mit Parserkombinatoren

Parser für Terminale

Parser für Terminale können mit den üblichen Verfahren für die Erkennung regulärer Sprachen realisiert werden.

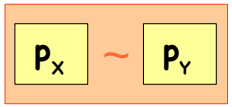
Parserkombinator für Produktionen

Parser für Produktionen werden von einem Kombinator $C_{\sim}$ für Produktionen erzeugt:

Ein Kombinator $C_{\sim}$ für Produktionen

konstruiert aus einem Parser p_X für X und einem Parser p_Y für Y einem Parser $p_{X\,Y}$ für X Y

$$C_{\sim}\,(p_X\,,\,p_Y) = p_{X\,Y}$$



p_{XY}

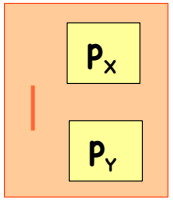
Parserkombinator für Nonterminale / Alternativen

Parser für Nichtterminale werden von einem Kombinator $C_{|}$ für Alternativen erzeugt:

Ein Kombinator $C_{|}$ für Alternativen

konstruiert aus einem Parser p_X für X und einem Parser p_Y für Y einen Parser $p_{X|Y}$ für Y | X

$$C_{|}\,(p_X\,,\,p_Y) = p_{X|Y}$$



$p_{X|Y}$

Funktionale Parser-Kombinatoren

Beispiel:

Angenommen wir haben Parser-Funktionen die „1“ und „2“ parsen können:

```
def parse_1(text: List[Char]) : Option[(Number, List[Char])] =
    if (text.head == '1') Some((Number(1), text.tail))
    else None

def parse_2(text: List[Char]) : Option[(Number, List[Char])] =
    if (text.head == '2') Some((Number(2), text.tail))
    else None
```

Die beiden Funktionen

- **akzeptieren einen Text – hier der Einfachheit halber eine Liste von Chars**
- **testen, ob es sich um das richtige Zeichen handelt und**
- **geben im Erfolgsfall eine erkannte Zahl und den Rest des Textes zurück.**

```
sealed abstract class ExpTree
case class Number(v: Int) extends ExpTree
```

Funktionale Parser-Kombinatoren

Beispiel:

Ein Parser, der „1 | 2“ parsen kann, ist auch leicht definiert:

```
def parse_1or2(text: List[Char]) : Option[(Number, List[Char])] =
  parse_1(text) match {
    case Some((num1, rest1)) => Some((num1, rest1))
    case None => parse_2(text)
}
```

Die Funktionen versucht „1“ zu parsen und wenn das fehlschlägt, dann versucht sie es mit „2“

Diese Konstruktion kann zu einem Alternativen-Kombinator verallgemeinert werden:

```
def parseAlt(
   parser_A: List[Char] => Option[(Number, List[Char])],
   parser_B: List[Char] => Option[(Number, List[Char])]): List[Char] => Option[(Number, List[Char])] = {
     (text: List[Char]) =>
          parse_A(text) match {
               case Some((num1, rest1)) => Some((num1, rest1))
               case None => parse_B(text)
          }
}
```

Die Funktionen nimmt zwei Parser die A bzw. B parsen können und macht daraus einen Parser der A|B parsen kann.

Objektorientierte Parser-Kombinatoren

Die Konstruktion kann auch objektorientiert gestaltet werden:

Wir definieren ein Trait (Interface) für Parser und zwei primitive Parser:

```
trait Parser {
  def parse(text: List[Char]) : Option[(Number, List[Char])]
}
```

Ein Parser-Interface

```
object Parse_1 extends Parser {
  def parse(text: List[Char]) : Option[(Number, List[Char])] =
     if (text.head == '1') Some((Number(1), text.tail))
     else None
}

object Parse_2 extends Parser {
  def parse(text: List[Char]) : Option[(Number, List[Char])] =
     if (text.head == '2') Some((Number(2), text.tail))
     else None
}
```

Ein Parser für „1“

Ein Parser für „2“

Objektorientierte Parser-Kombinatoren

Die Konstruktion eines Parsers für Alternativen ist jetzt eine Methode des Traits Parser:

```
trait Parser {

  def parse(text: List[Char]) : Option[(Number, List[Char])]

  def alt(other: Parser): Parser = {
    val outer = this
    new Parser{
        def parse(text: List[Char]): Option[(Number, List[Char])]  =
          outer.parse(text) match {
            case Some(result) => Some(result)
            case None         => other.parse(text)
          }
    }
  }
}
```

Ein Parserkombinator für Alternativen

```
println(Parse_1.alt(Parse_2).parse("2".toCharArray().toList))
```

```
Some((Number(2),List()))
```

Objektorientierte Parser-Kombinatoren

Mit einem | als Methodenname wird das Ganze noch etwas hübscher:

```
trait Parser {

  def parse(text: List[Char]) : Option[(Number, List[Char])]

  def |(other: Parser): Parser = {
    val outer = this
    new Parser{
        def parse(text: List[Char]): Option[(Number, List[Char])]  =
          outer.parse(text) match {
            case Some(result) => Some(result)
            case None         => other.parse(text)
          }
    }
  }
}
```

Ein Parserkombinator für |

```
println((Parse_1 | Parse_2).parse("2".toCharArray().toList))
```

```
Some((Number(2),List()))
```

Parser-Kombinatoren – Die Idee

Grammatik = Parser-Definition

Parser-Kombinatoren erlauben es „größere“ Parser aus „kleineren“ zusammen zu bauen

Mit moderner (funktionaler) Software-Technik können

Parser-Definitionen auf Basis von Parser-Kombinatoren

- **so gestaltet werden, dass sie wie Grammatik-Definitionen aussehen, bzw.**

Geeignete Grammatik-Defintionen

- **als Parser-Definitionen interpretiert werden**

Parser-Kombinatoren – Die Idee

Parser

die mit Parser-Kombinatoren konstruiert wurden, sind

- **Top-Down Parser,**
- **die nur Grammatiken ohne Links-Rekursion verarbeiten können, und**
- **mit Backtracking arbeiten**
  - **Alternativen werden „der Reihe nach“ abgearbeitet und eventuell abgebrochen**
  - **Solange bis eine erfolgreich ist**

Parser-Kombinatoren nutzen

- **Bequem und flexibel verwendbare Parser-Kombinatoren – speziell in einer typisierten Sprache – erfordern einen gewissen Softwaretechnischen Aufwand, auf den wir hier nicht eingehen wollen.**
- **Die Grundidee ist aber einfach und wir nutzen die Parser-Kombinatoren von Scala ohne auf technische Details einzugehen.**

Parser-Kombinatoren

Einfach und elegant, aber mit allen Nachteilen des Top-Down-Backtracking-Parsens:

- **unter Umständen exponentielle Laufzeit**
- **Linksrekursion nicht erlaubt**
- **Mehrdeutigkeit problematisch**

  Die erste Alternative wird verfolgt und kann – wenn erfolgreich geparst werden kann – dazu führen, dass alternative Regeln mit gleichem Anfang nicht in Betracht gezogen werden.

  Z.B. *Dangling Else* – Problem

exp ::= exp + exp | const

Linksrekursion

stmt ::= `if` ***expr*** `then` ***stmt***
`|` `if` ***expr*** `then` ***stmt*** `else` ***stmt***

Mehrdeutige Grammatik:
ein String hat mehr als eine passende Produktion

Parser-Kombinatoren

- **Idee von Philip Walder als Anwendungsbeispiel des funktionalen Programmierens Mitte der 1980er**
- **Konzept: *Top-Down-Backtracking* Parser**

  Implementierung einer Top-Down-Backtracking Parsing-Technik: Erkennungsfunktionen für jede Regel, mit Backtracking bei Misserfolg

PEG: Parsing Expression Grammar

Eine Grammatik definiert eine unendliche Menge
durch einen (nicht-deterministischen) Erzeugungsprozess

Ein Parser ist ein Erkennungs-Algorithmus

Grammatik => Parser

Definiere einen Erkennungsalgorithmus, der zum Erzeugungsprozess kompatibel ist

PEG:

- Definiere Grammatik und interpretiere sie als Erkennungsalgorithmus
- „Trick“ der Parsing mit Parser-Kombinatoren rechtfertigt:
  Statt: Grammatik => Sprache <=> Parser
  Jetzt: Grammatik = Sprache = Parser
- Von Bryan Ford definiert / popularisiert (siehe http://www.brynosaurus.com/pub/lang/peg.pdf)
- Basis / Rechtfertigung von Parser-Kombinatoren
  - Der Nichtdeterminismus wird als Reihenfolge der Versuche umgedeutet
  - Thema einer Veranstaltung zu Parsing / formalen Sprachen

Packrat-Parser

Backtrack-Parser

- Parser, die Alternativen untersuchen

  Backtracking: Suche ist Tiefensuche

  Problem: Suche kann sich in einem endlichen Unterraum verlieren, obwohl sie in einem anderen Unterraum erfolgreich wäre
- Suchstrategie ist relevant

  in der Regel von links nach rechts

  Grammatik: Reihenfolge der Alternativen beachten!

Prädikative Parser

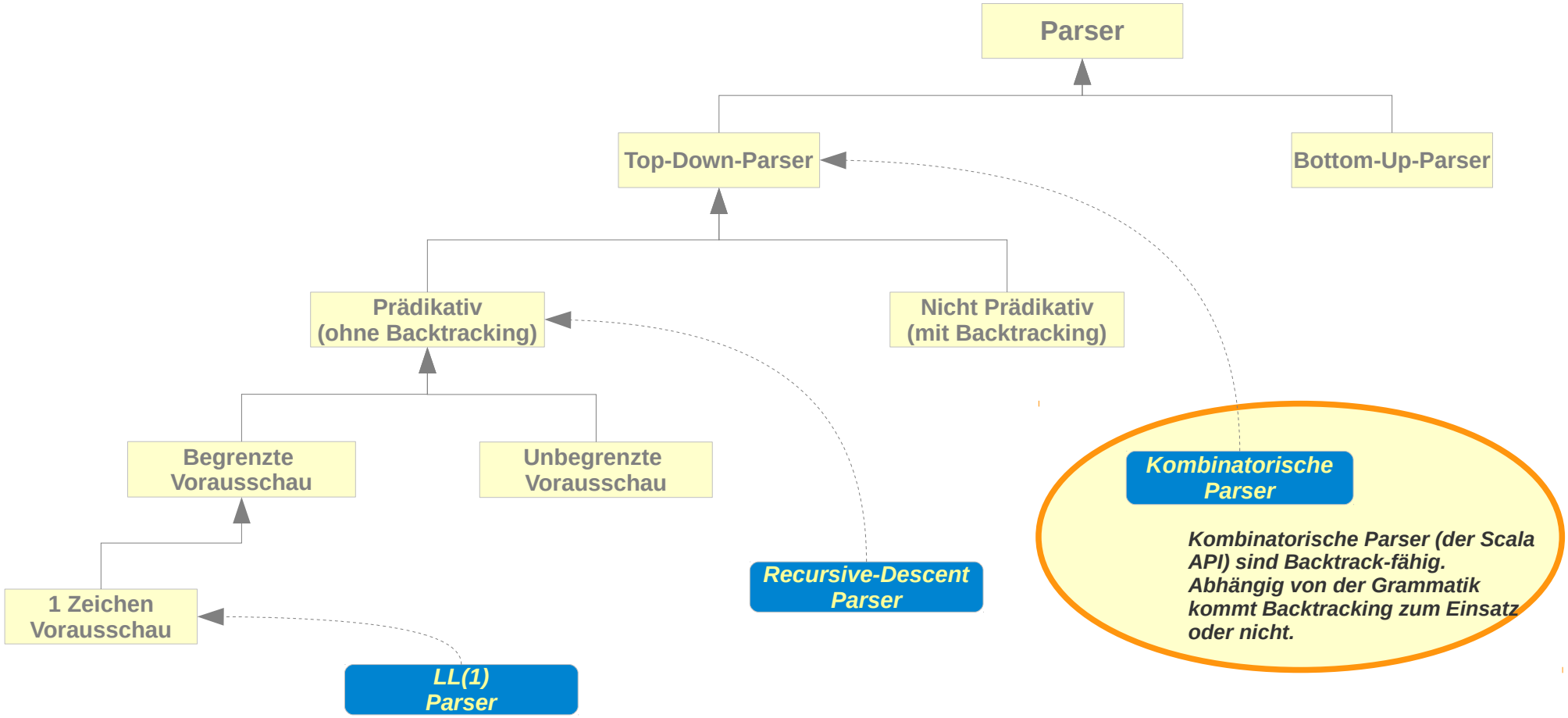
- Backtracking-freie Top-Down Parser

  Beispiel: Recursive-Descent-Parser

Packrat-Parser

- Parser der Ergebnisse „hamstert“ (Packrat: ein hamsterartiges kleines Pelztier)
- Merkt sich, welche (Teil-) Eingabe bereits geparst wurde
- Kommt darum mit links-rekursiven Grammatiken klar
- Thema einer Veranstaltung zu Parsing / formalen Sprachen

Parser – Überblick mit kombinatorischen Parsern



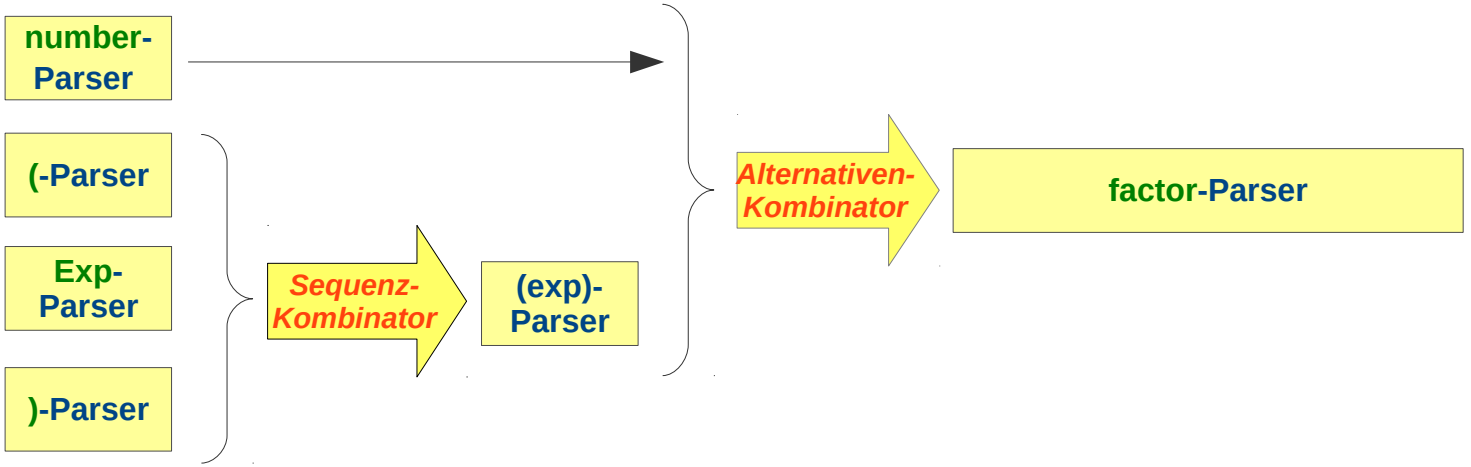
Parser-Kombinatoren in Scala

Beispiel

Grammatik-Fragment

factor ::= *number* | '(' *exp* ')'

Parser-Konstruktion



Parser-Kombinatoren in Scala

Beispiel

Beispiel aus Odersky, Spoon, Venners: „Programming in Scala“, 2nd edition artima

exp ::= *term* { ('+' | '-') *term* }
term ::= *factor* { ('*' | '/') *factor* }
factor ::= *number* | '(' *exp* ')'

Grammatik in EBNF-Form

Die Produktionen der Grammatik werden als Parser formuliert.

Die Parser erzeugen aus Strings die Ableitungsbäume entsprechend der (konkreten) Syntax.

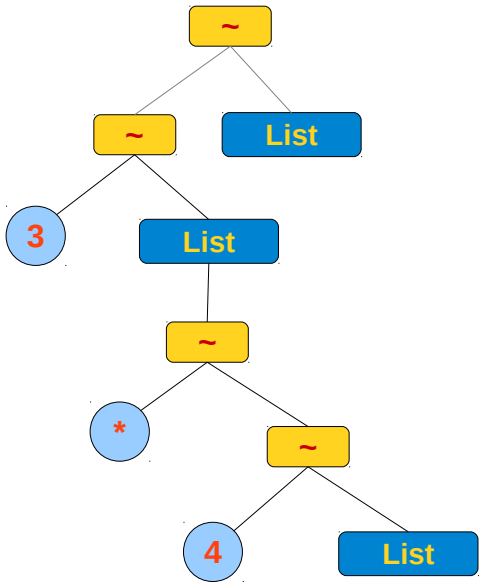
Parser für diese Grammatik

```
package parsing.ex_1

import scala.util.parsing.combinator.JavaTokenParsers

object FormulaParser extends JavaTokenParsers {

  def exp: Parser[Any] = term ~ rep(("+" | "-") ~ term )
  def term: Parser[Any] = factor ~ rep(("*" | "/") ~ factor )
  def factor: Parser[Any] = floatingPointNumber | "(" ~ exp ~ ")"

}

object SimpleFormula_App extends App {

  println(FormulaParser.parseAll(FormulaParser.exp, "3*4"))

}
```

*Ergebnis des Parsens von „3*4“:*
Ableitungsbaum . toString

```
[1.4] parsed: parsed:
((3~List((*~(4~List()))))~List())
```

```
(
 ( 3 ~
   List(
     ( * ~
       (4 ~ List())
     )
   )
 ) ~
 List()
)
```



Ableitungsbaum in Textform

Ableitungsbaum als Baum

Kombinatorisches Parsing

exp ::= *term* { ('+' | '-') *term* }

term ::= *factor* { ('*' | '/') *factor* }

factor ::= *number* | '(' *exp* ')'

EBNF-Form: implizite Operatoren

„ “ (nichts)	Sequenz (Aufeinanderfolge)
„{ ... }“	Wiederholung
„|“	Alternative

exp ::= *term* { ('+' | '-') *term* }

exp ist eine Sequenz
- aus einem *term*
- gefolgt von einer Wiederholung von
  - einer Sequenz von einer
    - Alternative von '+' und '-'
    - gefolgt von einem *term*

Für einen Parser für *exp*
benötigen wir Parser für
- *Terme,*
- „+“-Zeichen und
- „-“-Zeichen

und Parserkombinatoren für
- Sequenzen,
- Wiederholungen,
- Alternativen

Parser-Kombinatoren in Scala

- Parser-Kombinatoren sind eine Grundübung in funktionaler Programmierung
- Jede Sprache die „anständig funktional“ sein will, wird mit Parser-Kombinatoren „getestet“
- Scala hat Parser-Kombinatoren als Teil der API von Anfang an „offiziell“ unterstützt
- Um die Scala-API überschaubar zu halten wurde sie ab Scala 2.11 modularisiert: es ist nicht mehr alles „automatisch“ enthalten:
  - Kern (scala.lib) + eventuell weitere Module
- Parser-Kombinatoren nicht Bestandteil von scala.lib
- Die entsprechende jar-Datei muss explizit geladen und referenziert werden (Siehe Scala-Doku)

  beispielsweise so:
  - Lade / Installiere Scala
  - In Eclipse:

    Einstellungen: new User-Library *Parsing;*

    add external Jars : aus der Scala-Installation: *lib/scala-parser-combinators_... .jar*

    im Projekt: add User-Library *Parsing*

  oder in build.sbt so etwas wie:
  - libraryDependencies += "org.scala-lang.modules" %% "scala-parser-combinators" % "1.0.6"

Beispiel

Ein kombinatorischer Parser wird mit Hilfe von

- anderen Parsern und
- Parserkombinatoren

definiert.

Die Parserkombinatoren kombinieren Parser zu neuen Parsern

factor ::= *number* | '(' *exp* ')'

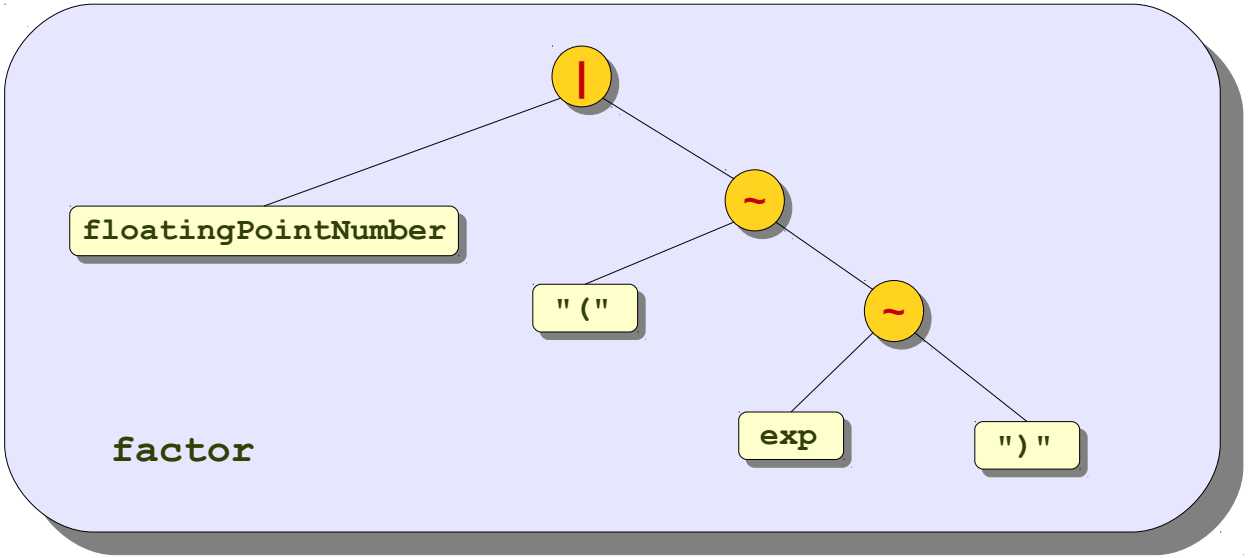
```
def factor: Parser[Any] = floatingPointNumber | "(" ~ exp ~ ")"
```

Genutzte Parser

- `floatingPointNumber`
- `exp`
- `"("`
- `")"`

Parserkombinatoren

- `|` Alternative
- `~` Verkettung



Beispiel

exp ::= *term* { ('+' | '-') *term* }

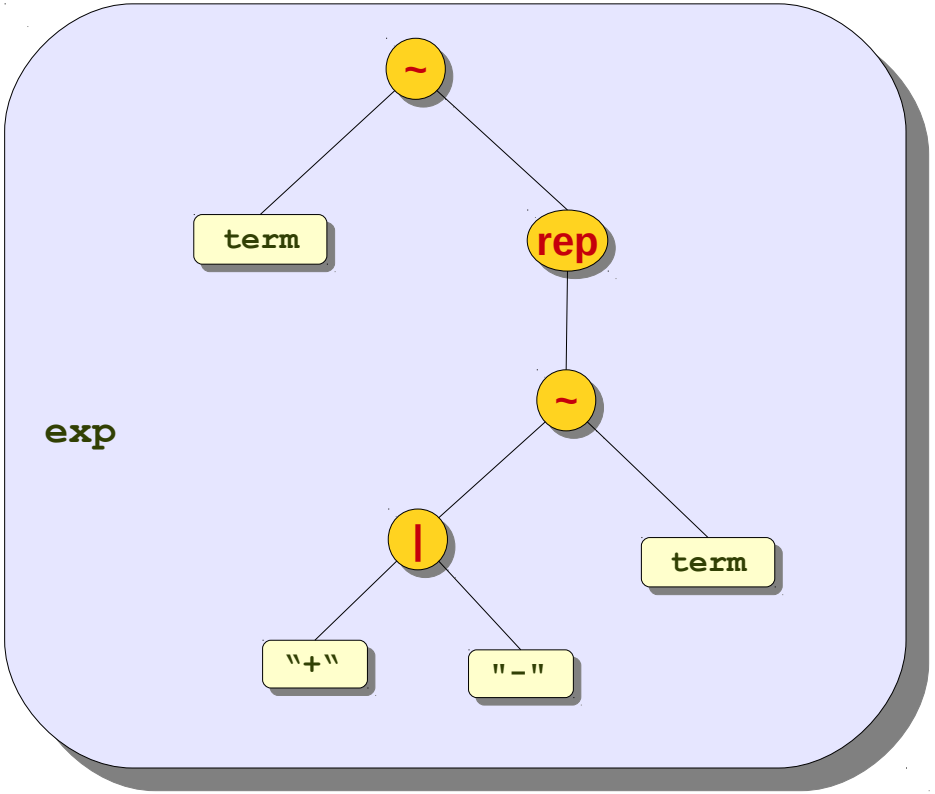
⟹ `def exp: Parser[Any] = term ~ rep(("+" | "-") ~ term )`

Genutzte Parser

- `term`
- `factor`
- `"("` *ein Scanner für* (
- `")"` *ein Scanner für*)

Parserkombinatoren

- `|` Alternative
- `~` Verkettung
- `rep` Wiederholung



Parser und ihre Nutzung

Für jede kontextfreie Grammatik in EBNF-Form kann so ein Parser definiert werden.

Nutzung

parseAll : Methode einer Parser-Klasse (*parst* die gesamte Eingabe)

Argumente: Parser, Eingabe

Ergebnis: konkreter Syntaxbaum

```
sealed abstract class ParseResult[+T] extends AnyRef
  A base class for parser results.

def  parseAll[T](p: Parser[T], in: Reader): ParseResult[T]
  Parse all of reader in with parser p.
```

Klasse definiert in scala.util.parsing.combinator.Parsers

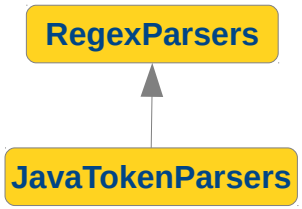
Methode definiert in scala.util.parsing.combinator.RegexParsers

RegexParsers

Reguläre Ausdrücke können als Parser ihrer Sprache verwendet werden

JavaTokenParsers

Vordefinierte reguläre Ausdrücke für „Java-Tokens“ können verwendet werden



Parser in der Scala API

JavaTokenParsers

Definiert reguläre Ausdrücke die Java-Tokens beschreiben

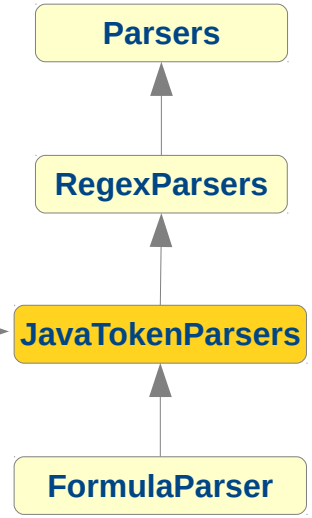
`JavaTokenParsers` *bietet Parser für folgende Tokenklassen:*

- `ident`
- `wholeNumber`
- `decimalNumber`
- `stringLiteral`
- `floatingPointNumber`

```
object FormulaParser extends JavaTokenParsers {
  def exp: Parser[Any] = term ~ rep(("+" | "-") ~ term )
  def term: Parser[Any] = factor ~ rep(("*" | "/") ~ term )
  def factor: Parser[Any] = floatingPointNumber | "(" ~ exp ~ ")"
}
```

```
def floatingPointNumber: Parser[String] =
    """-?(\d+(\.\d*)?|\d*\.\d+)([eE][+-]?\d+)?[fFdD]?""".r
```

Definiert in *scala.util.parsing.combinator.JavaTokenParsers*



Parser in der Scala API

JavaTokenParsers

```
import scala.util.parsing.combinator.JavaTokenParsers

trait ExpParser extends JavaTokenParsers  {
  def exp: Parser[Any]    = term~"+"~term | term
  def term: Parser[Any]   = factor~"*"~factor | factor
  def factor: Parser[Any] = decimalNumber | "("~exp~")"
}


object Test extends App with ExpParser {
  println(parseAll(exp, "(3 * 4) + 1 * 2"))
}
```

Parser

`exp` : Parser-Funktion für exp
`term` : Parser-Funktion für term
`factor`: Parser-Funktion für factor
`decimalNumber`: Scanner-Funktion für Zahlen
`"+"` : Scanner-Funktion für „+“
`"*"` : Scanner-Funktion für „*“

Parser-Kombinatoren

`~` : Verkettung
`|` : Alternative

Parser für die Grammatik

exp ::= term '+' term | term
term ::= factor * factor | factor
factor ::= decimalNumber | '(' exp ')'

`term~"+"~term` : Zwei Parser-Funktionen `term`, `"+"` werden mit mit dem Kombinator `~` zu einem neuen Parser verknüpft

Parser in der Scala API

Beispiele

```
import scala.util.parsing.combinator.JavaTokenParsers

trait ExpParser extends JavaTokenParsers  {
  def exp: Parser[Any] = term ~ rep("+" ~ term | "-"~term)
  def term: Parser[Any] = factor~rep("*"~factor | "/"~factor)
  def factor: Parser[Any] = decimalNumber | "("~exp~")"
}

object Test extends App with ExpParser_3 {
   println(parseAll(exp, "3 * 4 * (5 + 6)"))
}
```

Parser-Kombinatoren

~ : Verkettung

| : Alternative

rep : Wiederholung 0 oder mehrmals

Parser für die Grammatik

exp ::= term { '+' term | '-' term }*
term ::= factor { '*' factor | '/' factor}*
factor ::= decimalNumber | '(' exp ')'

Grammatik ist nicht links-rekursiv

```
import scala.util.parsing.combinator.JavaTokenParsers

trait ExpParser extends JavaTokenParsers {
  def exp : Parser[Any] = number ~ rep(op ~ number)
  def number = "[1-9]\\d".r
  def op     = "[\\|°]".r
}

object Test extends App with ExpParser_4 {
  println(parseAll(exp, "12 ° 13 | 20"))
}
```

Parser für die Grammatik

exp ::= number { op exp }*
op ::= '°' | '|'
number ::= *zweistellig ohne führende 0*

Grammatik nutzt Tokens die mit regulären Ausdrücken definiert werden

Parser in der Scala API

Beispiel

```
import scala.util.parsing.combinator.JavaTokenParsers

trait ExpParser_5 extends JavaTokenParsers {
  def exp: Parser[Any] = term ~ rep("+" ~ term | "-" ~ term)
  def term   = factor ~ rep("*" ~ factor | "/" ~ factor)
  def factor = wholeNumber | "(" ~ exp ~ ")" | call
  def call   = ident ~ "(" ~ repsep(exp, ",") ~ ")"
}

object Test extends App with ExpParser_5 {
  println(parseAll(exp, "f(5, g(6, 7)) + 8 * h(9)"))
}
```

Parser-Kombinatoren

`~` : Verkettung

`|` : Alternative

`rep` : Wiederholung 0 oder mehrmals

`repsep` : Wiederholung mit Trenner

Parser für die Grammatik

exp ::= term { '+' term | '-' term }*
term ::= factor { '*' factor | '/' factor}*
factor ::= wholeNumber | '(' exp ')' | call
call ::= idnet '(' param ')'
param::= exp | exp ',' param

Parser in der Scala API

JavaTokenParsers

Parserkombinatoren die im Trait JavaTokenParsers definiert oder ererbt werden

decimalNumber	eine Dezimalzahl
floatingPointNumber	eine Fließkommazahl (Dezimalzahl mit opt. Vorzeichen und E-Notation)
wholeNumber	ganze Zahl, eventuell mit Vorzeichen
ident	Bezeichner
literal(*String*)	Literal
stringLiteral	ein String in Hochkommas
opt(*P*)	optionales Vorkommen von *P*
rep(*P*)	wiederholtes Vorkommen von *P* (0 oder mehrmals)
rep1(*P*)	wiederholtes Vorkommen von *P* (mindestens einmal)
repN(*n*, *P*)	wiederholtes Vorkommen von *P* (genau *n* mal)
repsep(*P*, *Q*)	Wiederholungen von *P* die durch *Q* getrennt sind (0 oder mehrmals)
rep1sep(*P*, *Q*)	Wiederholungen von *P* die durch *Q* getrennt sind (mindestens einmal)
P* ~ *Q	Sequenz von *P* und *Q*
P* | *Q	Alternative *P* oder *Q*

Kombinatorisches Parsing : Interne DSL von Scala

Parserkombinatoren sind ein Beispiel für eine interne DSL

DSL : Domain Specific Language

Varianten einer DSL

- **externe DSL**
  DSL als eigenständige Sprache
  Programme werden von eigenständigen Tools verarbeitet
- **interne DSL**
  DSL als eingebettete Sprache: DSL-Konstrukte sind Teil der Wirts-Sprache
  Programme werden von den Tools der Wirt-Sprache verarbeitet

```
def ~ [U](q: => Parser[U]): Parser[~[T, U]]
```

Parserkombinatoren sind kein Sprachbestandteil, sondern „ganz normale“ Methoden.

Kombinator ~:
Methode eines Parsers mit Argument vom Typ Parser, liefert Ergebnis vom Typ Parser;
definiert in *scala.util.parsing.combinator.Parsers.Parser*

Parser

Ein Parser – definiert im Trait Parsers – ist eine Funktion, die

- **eine Eingabe (teilweise) konsumiert und**
- **ein Parsing-Ergebnis liefert**
- **das Parsing-Ergebnis beinhaltet einen Eingabe-Rest**

```
trait Parsers {

  type Elem

  type Input = Reader[Elem]

  sealed abstract class ParseResult[+T] { ... }

  case class Success[+T](result: T, override val next: Input) extends ParseResult[T] { ... }
  sealed abstract class NoSuccess(val msg: String, override val next: Input) extends ParseResult[Nothing] { ... }

  abstract class Parser[+T] extends (Input => ParseResult[T]) { ... }

}
```

Siehe auch:

- API-Doku: https://static.javadoc.io/org.scala-lang.modules/scala-parser-combinators_2.12/1.0.6/index.html
- A. Moors, F. Piessens, M. Odersky: *Parser Combinators in Scala*
  http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.142.5002&rep=rep1&type=pdf

Parser

Typ-Parameter T

- **Typ des Ergebnisses des Parsings**
- **in der Regel ein AST (Parser im üblichen Sinn) oder eine Sequenz von Tokens (Scanner)**

```
trait Parsers {

  type Elem

  type Input = Reader[Elem]

  sealed abstract class ParseResult[+T] { ... }

  /** The success case of `ParseResult`: contains the result and the remaining input.
   *
   *  @param result The parser's output
   *  @param next   The parser's remaining input
   */
  case class Success[+T](result: T, override val next: Input) extends ParseResult[T] { ...
}
  sealed abstract class NoSuccess(val msg: String, override val next: Input) extends ParseResult[Nothing] { ... }

  abstract class Parser[+T] extends (Input => ParseResult[T]) { ... }

}
```

Parser: Token-Parser und Character-Parser

Typ Elem

- Typ der Elemente, aus denen die Eingabe besteht
- Scanner und Parser werden nicht an Hand der Eingabe unterschieden
- Scanner und Parser werden nicht an Hand der Grammatik unterschieden: Parser können reguläre Sprachen erkennen

```
trait Parsers {

 /** the type of input elements the provided parsers consume (When consuming
   *  individual characters, a parser is typically called a ''scanner'', which
   *  produces ''tokens'' that are consumed by what is normally called a ''parser''.
   *  Nonetheless, the same principles apply, regardless of the input type.) */
  type Elem

  type Input = Reader[Elem]

  sealed abstract class ParseResult[+T] { ... }

  case class Success[+T](result: T, override val next: Input) extends ParseResult[T] { ... }
  sealed abstract class NoSuccess(val msg: String, override val next: Input) extends ParseResult[Nothing] { ... }

  abstract class Parser[+T] extends (Input => ParseResult[T]) { ... }

}
```

Character-Parser

RegExParsers

- Parser für einen Input, der aus Chars besteht
- Strings und Reguläre Ausdrücke können als Parser genutzt werden
- Unterklasse JavaTokenParsers definiert reguläre Ausdrücke für die typischen Java-Tokens

```
trait RegexParsers extends Parsers {
  type Elem = Char
  ...
}

trait JavaTokenParsers extends RegexParsers {
  ...
}
```

Token-Parser

TokenParsers

- Parser für einen Input, der aus Tokens besteht
- Unterklasse StdTokenParsers definiert Parser für Standard-Tokens (numerische Literale, Bezeichner, Strings)

```
trait TokenParsers extends Parsers {
  type Tokens <: token.Tokens
  ...
}

trait StdTokenParsers extends TokenParsers {
  type Tokens <: StdTokens
  ...
}
```

Konkrete => abstrakte Syntax

Definition des AST

```
// Sorte Exp
sealed abstract class Exp

// Operatoren
case class Plus(l: Exp, r: Exp) extends Exp
case class Times(l: Exp, r: Exp) extends Exp
case class Const(n: Int) extends Exp
```

Case-Klassen

eine Variante der Klassendefinition, die eingesetzt wird für Klassen
- deren Exemplare (typischerweise) unveränderlich sind
- deren Exemplare mit Pattern-Matching verarbeitet werden sollen

Case-Classes bieten
- generierte Methoden equals und hashcode
- generierte Getter und Setter für die Konstruktor-Parameter
- eine generierte toString-Methode

Case-Classes ~ algebraische Datentypen
- eignen sich zur Definition von algebraischen Datentypen (also von *geordneten Bäume mit typisierten Knotenkonstruktoren*)

Sealed Classes

dürfen nur in der Quelldatei abgeleitet werden, in der sie definiert sind.
Verbessert Pattern-Matching: alle Möglichkeiten sind fix und dem Compiler bekannt

Konkrete => abstrakte Syntax

Konstruktion des AST

Beispiel: Ausdrücke parsen und einen AST vom Typ Exp erzeugen

```
import scala.util.parsing.combinator.JavaTokenParsers

// Sorte Exp
sealed abstract class Exp

// Operatoren
case class Const(n: Int) extends Exp
case class Plus(l: Exp, r: Exp) extends Exp
case class Times(l: Exp, r: Exp) extends Exp


object ExpParser extends JavaTokenParsers {

  def exp: Parser[Exp] = decimalNumber     ^^ { d => Const(d.toInt) } |
                         plus          |
                         times

  def plus: Parser[Exp] = "("~exp~"+"~exp~")"  ^^ { case "(" ~ e1 ~ "+" ~ e2 ~ ")" => Plus(e1, e2)}

  def times: Parser[Exp] = "("~exp~"*"~exp~")" ^^ { case "(" ~ e1 ~ "*" ~ e2 ~ ")" => Times(e1, e2)}
}



object AstBuilding_App extends App {
   println(ExpParser.parseAll(ExpParser.exp, "((2 + 3) * (4 + 5))"))
}
```

```
[1.20] parsed:
Times(Plus(Const(2),Const(3)),Plus(Const(4),Const(5)))
```

Konkrete => abstrakte Syntax

^^ Kombinator: Transformation von Parsern

```
/** A parser combinator for function application.
 *
 *  `p ^^ f` succeeds if `p` succeeds; it returns `f` applied to the result of `p`.
 *
 * @param f a function that will be applied to this parser's result.
 * @return a parser that has the same behaviour as the current parser, but whose result is
 *         transformed by `f`.
 */
def ^^ [U](f: T => U): Parser[U] = ...
```

Der ^^ Kombinator

wird in *scala.util.parsing.combinator.Parsers.Parser* als (Operator-) Methode definiert und ist ein Transformator für Parsing-Ergebnisse.

Ein Parser der T-Objekte erzeugt wird durch die Methode ^^ mit einer Funktion, die T-Ojekte in U-Objekte abbildet, transformiert zu einem Parser der U-Objekte erzeugt

Parser[T]* . ^^(*T* => *U*) erzeugt einen *Parser[U]

Konkrete => abstrakte Syntax

^^ Kombinator

Beispiel

```
decimalNumber    ^^ { d => Const(d.toInt) }
```

Funktion: d => Const(d.*toInt*)

Wird genutzt in

```
def exp: Parser[Exp] = decimalNumber    ^^ { d => Const(d.toInt) }
```

In scala.util.parsing.combinator.JavaTokenParsers ist definiert:

def decimalNumber: Parser[String]

Es ist ein Parser der Strings erzeugt,

die Funktion `{ d => Const(d.toInt) }` macht aus einem String ein Exp-Objekt,

als Argument der Methode ^^ wird aus einem Parser der Strings erzeugt, ein Parser der ein Exp erzeugt.

Parser[String] ^^(String => Exp) = Parser[Exp]

Konkrete => abstrakte Syntax

^^ Kombinator

Beispiel

```
"("~exp~"+"~exp~")" ^^ { case "(" ~ e1 ~ "+" ~ e2 ~ ")"
                         => Plus(e1, e2) }
```

Funktionsdefinition via Pattern-Match:

Das Parsing-Ergebnis wird mit

```
"(" ~ e1 ~ "+" ~ e2 ~ ")"
```

verglichen und dabei werden die Variablen e1 und e2 an die „gematchten“ Werte gebunden.

```
def fun_0 (x : AorB): Unit = x match {
  case A => println("an A")
  case B => println("an B")
}
 <==>
val fun_1: AorB => Unit = {
  case A => { println("an A") }
  case B => { println("an B") }
}
```

Etwas Scala-Magie: Kurz-Notation für Funktionen mit einem Argument, die aus einem Match auf diesem Argument bestehen.

Konkrete => abstrakte Syntax

Vereinfachungen mit ~>, <~

```
def plus: Parser[Exp] =  "("~exp~"+"~exp~")" ^^ { case "(" ~ e1 ~ "+" ~ e2 ~ ")"
                                                    => Plus(e1, e2) }
```

```
def plus: Parser[Exp] = ("(" ~> exp <~ "+") ~ exp <~")" ^^ { case e1 ~ e2
                                                             => Plus(e1, e2)}
```

P **~>** ***Q*** **: Das Gesamtergebnis ist das von Q (parse und dann vergiss *P*)**

P **<~** ***Q*** **: Das Gesamtergebnis ist das von P (parse und dann vergiss *Q*)**

Listen parsen

Übernahme von Listen in den AST

```
// AST
sealed abstract class Exp
case class Const(n: Int) extends Exp
case class Call(fun: Symbol, args: List[Exp]) extends Exp
```

```
// konkrete Syntax
trait ExpParser extends JavaTokenParsers {
  def exp: Parser[Any]     = wholeNumber | call
  def call: Parser[Any]    = ident ~ "(" ~ repsep(exp, ",") ~ ")"
}
```

```
// konkrete Syntax mit AST-Konstruktion
trait ExpParser extends JavaTokenParsers {

  def exp: Parser[Exp]  = wholeNumber ^^ { d => Const(d.toInt) } |
                          call


  def call: Parser[Exp]    = ident ~ ("(" ~> repsep(exp, ",")) <~ ")" ^^
                                    { case id~args => Call(Symbol(id), args) }
}
```

Konkrete => abstrakte Syntax

Listen parsen

```
// AST
sealed abstract class Exp
case class Const(n: Int) extends Exp
case class Plus(l: Exp, r: Exp) extends Exp
case class Plus(l: Exp, r: Exp) extends Exp
case class Times(l: Exp, r: Exp) extends Exp
case class Div(l: Exp, r: Exp) extends Exp
```

```
// Konkrete Syntax
trait ExpParser_4 extends JavaTokenParsers {
  def exp: Parser[Any] = repsep(term, addOp)
  def addOp = "+" | "-"
  def term = repsep(factor, multOp)
  def multOp = "*" | "/"
  def factor = wholeNumber | "("~exp~")"
}
```

Probleme bei der Baumkonstruktion:

Die Struktur der konkreten Syntax unterscheidet sich erheblich von der der abstrakten Syntax.

repsep wirft sein zweites Argument (i.d.R. ist dies das Trennzeichen) weg, das „Trennzeichen“ (+, -, *, /) ist hier aber wesentlich für den Baumaufbau.

Konkrete => abstrakte Syntax

Listen parsen

^^^ und chainl1

```
trait ExpParser extends JavaTokenParsers {

  def exp: Parser[Exp]
          = chainl1(term, term, addOp)

  def addOp : Parser[(Exp, Exp) ⇒ Exp]
          = "+" ^^^ {(x:Exp, y: Exp) => Plus(x,y)} |
            "-" ^^^ {(x:Exp, y: Exp) => Minus(x,y)}

  def term = chainl1(factor, factor, multOp)

  def multOp : Parser[(Exp, Exp) ⇒ Exp]
          = "*" ^^^ {(x:Exp, y: Exp) => Times(x,y)}  |
            "/" ^^^ {(x:Exp, y: Exp) => Div(x,y)}

  def factor: Parser[Exp]
          = wholeNumber   ^^ { d => Const(d.toInt) } |
            "("~>exp<~")" ^^ { e => e }
}


object Test extends App with ExpParser {
  println(parseAll(exp, "2 * (3 + 4) - 5 / 6"))
}
```

^^^ ist eine Variante von ^^:
^^ Funktion; ^^^ Konstante
^^^ c ≃ { x => c }

chainl1
entspricht repsep1 und hat (hier) drei Argumente:
- **P1 = Parser für der erste Listenelement**
- **P2 = Parser für weitere Listenelemente**
- **TP = Parser für den Trenner**

Der Parser für den Trenner muss eine Funktion liefern, die (abhängig vom Trennsymbol) eine „Zusammenbau“-Funktion liefern.

`chainl1(term, term, addOp)` **Parse eine Liste mit**
- `term` **als Parser für das erste Element**
- `term` **als Parser für weitere Elemente**
- `addOp` **als Funktion die Zwischenergebnisse verarbeitet**

Vollständiges Beispiel

```
object TestExpParser extends App {
   println(
     ExpParser.parse(
        "2 + 3*(3+5)/2 - 2/3 "))
}
```

phrase
generiert einen Parser der den Input vollständig abarbeitet und auf Eingabeströme angewendet werden kann.

```
import scala.util.parsing.combinator.JavaTokenParsers
import scala.util.parsing.input.CharSequenceReader

// abstract Syntax Tree
sealed abstract class Exp
case class Const(n: Int) extends Exp
case class Plus(l: Exp, r: Exp) extends Exp
case class Minus(l: Exp, r: Exp) extends Exp
case class Times(l: Exp, r: Exp) extends Exp
case class Div(l: Exp, r: Exp) extends Exp

// Parser
object ExpParser extends JavaTokenParsers {

  private def exp: Parser[Exp]
            = chainl1(term, term, addOp)

  private def addOp : Parser[(Exp, Exp) ⇒ Exp]
            = "+" ^^^ {(x:Exp, y: Exp) => Plus(x,y)} |
              "-" ^^^ {(x:Exp, y: Exp) => Minus(x,y)}

  private def term = chainl1(factor, factor, multOp)

  private def multOp : Parser[(Exp, Exp) ⇒ Exp]
                     = "*" ^^^ {(x:Exp, y: Exp) => Times(x,y)}  |
                       "/" ^^^ {(x:Exp, y: Exp) => Div(x,y)}

  private def factor: Parser[Exp]
            = wholeNumber   ^^ { d => Const(d.toInt) } |
              "("~>exp<~")" ^^ { e => e }

  def parse(s: String) : Exp =
    phrase(exp)(new CharSequenceReader(s)) match {
        case Success(e,_)     => e
        case NoSuccess(msg,_) => throw new IllegalArgumentException(
                                    "Parser Error  '" + s + "': " + msg)
    }
}
```

Parser-Kombinatoren

Dangling Else – Problem:

```
[1.22] failure: string matching regex `\z' expected but `e' found

if (x>y) then a := 5 else a := 6
                         ^
```

\z : Eingabe-Ende

```
import scala.util.parsing.combinator.JavaTokenParsers

object IfThenElseParser extends JavaTokenParsers {

  def stmt: Parser[Any] =
    "if" ~ "(" ~ exp ~ ")" ~ "then" ~ stmt |
    "if" ~ "(" ~ exp ~ ")" ~ "then" ~ stmt ~ "else" ~ stmt |
    ident ~ ":=" ~ exp

  def exp: Parser[Any] = term ~ rep(("<" | ">") ~ term )
  def term: Parser[Any] = factor ~ rep(("*" | "/" | "*" | "/") ~ term )
  def factor: Parser[Any] = floatingPointNumber | ident | "(" ~ exp ~ ")"

}

object DanglingElseApp extends App {
   println(IfThenElseParser.parseAll(IfThenElseParser.stmt, "if (x>y) then a := 5 else a := 6"))
}
```

```
[1.33] parsed:
(((((((if~()~((x~List())~List((>~(y~List()))))))~))~then)~((a
~:=)~((5~List())~List())))~else)~((a~:=)~((6~List())~List())
))
```

```
import scala.util.parsing.combinator.JavaTokenParsers

object IfThenElseParser extends JavaTokenParsers {

  def stmt: Parser[Any] =
    "if" ~ "(" ~ exp ~ ")" ~ "then" ~ stmt ~ "else" ~ stmt |
    "if" ~ "(" ~ exp ~ ")" ~ "then" ~ stmt |
    ident ~ ":=" ~ exp

  def exp: Parser[Any] = term ~ rep(("<" | ">") ~ term )
  def term: Parser[Any] = factor ~ rep(("*" | "/" | "*" | "/") ~ term )
  def factor: Parser[Any] = floatingPointNumber | ident | "(" ~ exp ~ ")"

}

object DanglingElseApp extends App {
   println(IfThenElseParser.parseAll(IfThenElseParser.stmt, "if (x>y) then a := 5 else a := 6"))
}
```

Reihenfolge ist wichtig

Packrat-Parser

PEG: *Parsing Expression Grammar* (Bryan Ford in 2004)

Eine Alternative zu kontextfreien Grammatiken um die Syntax formaler Sprachen zu definieren

Wesentliches Konzept:

- Philosphisch: Statt Produktionsregeln werden Erkennungsregeln definiert
- Praktisch: Kein Unterschied zu einer kontextfreien Grammatik, außer dass bei Alternativen die erste passende Alternative den Text erkennt

stmt ::= `if` *expr* `then` *stmt*
| `if` *expr* `then` *stmt* `else` *stmt*

Diese Grammatik ist **nicht uneindeutig** in Bezug auf

if x>y then x =1 else x=2

Dieser String ist schlicht **FALSCH** (es sei denn es gibt eine mit else beginnende Produktion): Die erste Regel passt und wird erkannt.

Packrat-Parser

Parser die PEGs parsen und dabei einmal erkannte Teilstrings speichern (Packrats „hamstern“ Futter)

Dieses Vorgehen

- erhöht den Speicherbedarf,
- beschleunigt aber das Parsen
- Und ermöglicht Linksrekursion: Die gleiche Produktion wird nicht zweimal an der gleichen Stelle im String ausprobiert.

Packrat (Neotoma cinerea)
Bildquelle: Wikipedia

Packrat-Parser

Beispiel:

```
import scala.util.parsing.combinator.JavaTokenParsers
import scala.util.parsing.combinator.PackratParsers

object PRParser extends JavaTokenParsers with PackratParsers {

  lazy val term: PackratParser[Any] =
    term ~ "+" ~ term |
    term ~ "-" ~ term |
    factor

   lazy val factor: PackratParser[Any] =
    factor ~ "*" ~ factor |
    factor ~ "/" ~ factor |
    "(" ~ term ~ ")" |
    floatingPointNumber
}

object Packrat_App extends App {
  println(PRParser.parseAll(PRParser.term, "1+3*4"))
}
```

```
[1.6] parsed: ((1~+)~((3~*)~4))
```

```
object NonPRParser extends JavaTokenParsers {

  lazy val term: Parser[Any] =
    term ~ "+" ~ term | term ~ "-" ~ term |  factor

   lazy val factor: Parser[Any] =
    factor ~ "*" ~ factor | factor ~ "/" ~ factor | "(" ~ term ~ ")" | floatingPointNumber

}

object Packrat_App extends App {
  println(NonPRParser.parseAll(NonPRParser.term, "1+3*4"))
}
```

```
Exception in thread "main"
java.lang.StackOverflowError
```

Übersicht: Parserkombinatoren in Scala

Klassenübersicht

Parsers

Basisklasse für Parser

RegexParsers

Basisklasse für Parser die Strings parsen

bietet implizite Konversion von einem RegEx zu einer Parser der erkannte Strings parst

JavaTokenParsers

Subklasse von RegExParsers (!) kennt die RegExe für Java Tokens

TokenParsers

Basisklasse für Parser, die Tokens verarbeiten

StandardTokenParsers

Basisklasse für Parser, die vordefinierte (Standard-) Tokens verarbeiten

JavaTokenParsers

werden verwendet wenn die lexikalischen Elemente denen von Java entsprechen
Beispiel:

```
object ExpParser extends JavaTokenParsers {

  private def exp: Parser[Exp] = chainl1(term, term, addOp)

  private def addOp : Parser[(Exp, Exp) ⇒ Exp] =
    "+" ^^^ {(x:Exp, y: Exp) => Plus(x,y)} |
    "-" ^^^ {(x:Exp, y: Exp) => Minus(x,y)}

  private def term = chainl1(factor, factor, multOp)

  private def multOp : Parser[(Exp, Exp) ⇒ Exp] =
    "*" ^^^ {(x:Exp, y: Exp) => Mult(x,y)} |
    "/" ^^^ {(x:Exp, y: Exp) => Div(x,y)}

  // wholeNumber is a regex defined in JavaTokenParsers
  private def factor: Parser[Exp] =
    wholeNumber   ^^ { d => Const(d.toInt) } |
    "("~>exp<~")" ^^ { e => e }

  def parse(s: String) : Exp =
    phrase(exp)(new CharSequenceReader(s)) match {
      case Success(e,_)     => e
      case NoSuccess(msg,_) => throw new IllegalArgumentException(
        "Parser Error  '" + s + "': " + msg)
    }
}
```

RegExParsers

werden verwendet wenn die lexikalischen Elemente nicht denen von Java entsprechen
Beispiel:

```
object ExpParser extends RegexParsers {

  // regex definition
  def wholeNumberPattern = """\d+""".r


  private def exp: Parser[Exp] = chainl1(term, term, addOp)

  private def addOp : Parser[(Exp, Exp) ⇒ Exp] =
    "+" ^^^ {(x:Exp, y: Exp) => Plus(x,y)} |
      "-" ^^^ {(x:Exp, y: Exp) => Minus(x,y)}

  private def term = chainl1(factor, factor, multOp)

  private def multOp : Parser[(Exp, Exp) ⇒ Exp] =
    "*" ^^^ {(x:Exp, y: Exp) => Mult(x,y)} |
      "/" ^^^ {(x:Exp, y: Exp) => Div(x,y)}

  // a regex may be used as parser
  private def factor: Parser[Exp] =
    wholeNumberPattern   ^^ { d => Const(d.toInt) } |
      "("~>exp<~")" ^^ { e => e }

  def parse(str: String) : Exp = parseAll(exp, str) match {
    case Success(result, _) => result
    case failure : NoSuccess => scala.sys.error(failure.msg)
  }
}
```

Übersicht: Parserkombinatoren in Scala

StandardTokenParsers

werden verwendet wenn die lexikalischen Elemente neu definiert werden sollen.

Beispiel:

```
object ProgParser extends StandardTokenParsers {

  // definition of keywords in value lexical of StandardTokenParsers
  lexical.reserved += ("program", "var", "value", "read", "end")

  // definition of delimiters in value lexical of StandardTokenParsers
  lexical.delimiters += ("+", "-", "*", "/", ":=", "(", ")", ";" )

  private def exp: Parser[Exp] = chainl1(term, term, addOp)

  private def addOp : Parser[(Exp, Exp) ⇒ Exp] =
    "+" ^^^ {(x:Exp, y: Exp) => Plus(x,y)} |
      "-" ^^^ {(x:Exp, y: Exp) => Minus(x,y)}

  private def term = chainl1(factor, factor, multOp)

  private def multOp : Parser[(Exp, Exp) ⇒ Exp] =
    "*" ^^^ {(x:Exp, y: Exp) => Mult(x,y)} |
      "/" ^^^ {(x:Exp, y: Exp) => Div(x,y)}

  // numericLit is defined in StandardTokenParsers as a parser that parses numeric literals (to strings)
  // numeric literals are tokens that are formed by sequences of digits
  private def factor: Parser[Exp] =
    numericLit                    ^^ { digits => Const(digits.toInt) } |
    ident                         ^^ { str    =>  Var(str) }           |
    "read"                        ^^ { name => Read }                  |
    "(" ~> exp <~ ")"             ^^ { e => e }

  def definition: Parser[Definition] =
    ("var" ~> ident <~ ":=") ~ (exp <~ ";")  ^^ { case name ~ value => Definition(name, value) }

  def program: Parser[Program] =
    ("program" ~> rep(definition)) ~ ("value" ~> exp <~ "end") ^^ { case defs ~ value => Program(defs, value) }

  // StandardTokenParsers do not have a parseAll method. parseAll is defined in RegexParsers,
  // use phrase(program) on the tokens created by the lexer
  def parse(str: String) : Program = {
    val lexer = new lexical.Scanner(str)  // lexical is defined in StandardTokenParsers as val lexical = new StdLexical
    phrase(program)(lexer) match {
      case Success(tree,_)     => tree
      case m@NoSuccess(msg,_)  => println(m)
        throw new IllegalArgumentException("Parser Error")
    }
  }
}
```

Übersicht: Parserkombinatoren in Scala

TokenParsers

werden verwendet wenn die lexikalische und syntaktische Ebene getrennt werden soll (Nur für Fortgeschrittene !)
Beispiel lexikalische Ebene / Tokens definieren:

```
import scala.util.parsing.combinator.token.Tokens

trait ExpTokens extends Tokens {

  sealed abstract class ExpToken extends Token

  case class NumberToken(chars: String) extends ExpToken {
    override def toString: String = "NUM_T(" + chars + ")"
  }

  case class OperatorToken(chars: String) extends ExpToken {
    override def toString: String = "OP_T(" + chars + ")"
  }
  case class LeftPToken(chars: String) extends ExpToken {
    override def toString: String = "(_T"
  }
  case class RightPToken(chars: String) extends ExpToken {
    override def toString: String = ")_T"
  }

}
```

Übersicht: Parserkombinatoren in Scala

TokenParsers

Beispiel lexikalische Ebene:

```
class ExpLexical extends RegexParsers with ExpTokens {

  // define lexical structure of the tokens as regular expressions
  private val numberPatS = """(0|(?:[1-9][0-9]*))"""
  private val operatorPatS = """(\+|\-|\*|/)"""
  private val leftPPatS = """(\()"""
  private val rightPPatS = """(\))"""

  private val NumberPat   = numberPatS.r
  private val OperatorPat = operatorPatS.r
  private val LeftPPat    = leftPPatS.r
  private val RightPPat   = rightPPatS.r

  private val whiteSpacePat = """\s+""".r

  /** A parser that produces a token (from a stream of characters). */
  private def token: Parser[ExpToken] =
    NumberPat      ^^ { str => NumberToken(str) }      |
    OperatorPat    ^^ { str => OperatorToken(str) }    |
    LeftPPat       ^^ { str => LeftPToken(str) }       |
    RightPPat      ^^ { str => RightPToken(str) }      |
    failure("illegal character")

  /** A parser that produces a list of tokens (from a stream of characters). */
  private def tokens: Parser[List[ExpToken]] = rep(token)

  /** A function that produces tokens form a string. */
  def tokenize(code: String): List[ExpToken] = {
      parse(tokens, code) match {
        case NoSuccess(msg, next) => throw new Exception(msg)
        case Success(result, next) => result
      }
  }


}
```

```
// A scanner is just a reader for tokens
class Scanner(src: String) extends Reader[ExpToken] {
  import scala.util.parsing.input.{Position, NoPosition}

  val tokenList = tokenize(src)
  var p = 0

  override def first: ExpToken = tokenList(p)

  override def rest: Reader[ExpToken] = {
    p = p +1
    this
  }
  override def atEnd: Boolean = (p == tokenList.length)
  def pos: Position = NoPosition
}
```

Übersicht: Parserkombinatoren in Scala

TokenParsers Beispiel syntaktische Ebene

```
object ExpParser extends TokenParsers {

  val lexical = new ExpLexical

  import lexical.{ExpToken, OperatorToken, NumberToken, LeftPToken, RightPToken}


  override type Tokens = ExpTokens

  override type Elem = lexical.Token

  // Conversion to turn tokens into parsers automatically is provided by TokenParsers
  //implicit def tokenToParser(tok: ExpToken): Parser[ExpToken] = ...

  // parse number token
  // cumbersome because the is no predefined parser for classes(!) instead for objects as above.
  val number : Parser[lexical.Token] =
     elem("number", _.isInstanceOf[NumberToken])

  private def exp: Parser[ExpTree] = chainl1(term, term, addOp)

  private def addOp : Parser[(ExpTree, ExpTree) => ExpTree] =
    OperatorToken("+") ^^^ {(x:ExpTree, y: ExpTree) => Operation(x, Operator('+'),  y)} |
    OperatorToken("-") ^^^ {(x:ExpTree, y: ExpTree) => Operation(x, Operator('-'),  y)}

  private def term = chainl1(factor, factor, multOp)

  private def multOp : Parser[(ExpTree, ExpTree) => ExpTree] =
    OperatorToken("*") ^^^ {(x:ExpTree, y: ExpTree) => Operation(x, Operator('*'),  y)} |
    OperatorToken("/") ^^^ {(x:ExpTree, y: ExpTree) => Operation(x, Operator('/'),  y)}

  private def factor: Parser[ExpTree] =
    LeftPToken("(") ~> exp <~ RightPToken(")") |
    number           ^^ { (n: lexical.Token)   => Number(n.chars.toInt) }


}
```

```
def parse(code: String) : ExpTree = {
  val scanner = new lexical.Scanner(code)

  phrase(exp)(scanner) match {
    case Success(tree,_)       => tree
    case error@NoSuccess(_,_)  => println(error)
      throw new IllegalArgumentException("Parser Error")
  }
}
```

Quellen

- **API http://www.scala-lang.org/api/2.11.8/scala-parser-combinators/**
- **M. Odersky, L. Spoon, B. Venners: *Programming in Scala*, 3nd edition**
- ***Parsing expression grammar:***
  http://en.wikipedia.org/wiki/Parsing_expression_grammar
  http://bford.info/pub/lang/peg-slides/
  http://bford.info/pub/lang/thesis/
- **E. Labun: *Combinator Parsing in Scala,* Masterarbeit THM: http://labun.com/fh/ma.pdf**