Compilerbau CS1019

Th. Letschert

TH Mittelhessen Gießen

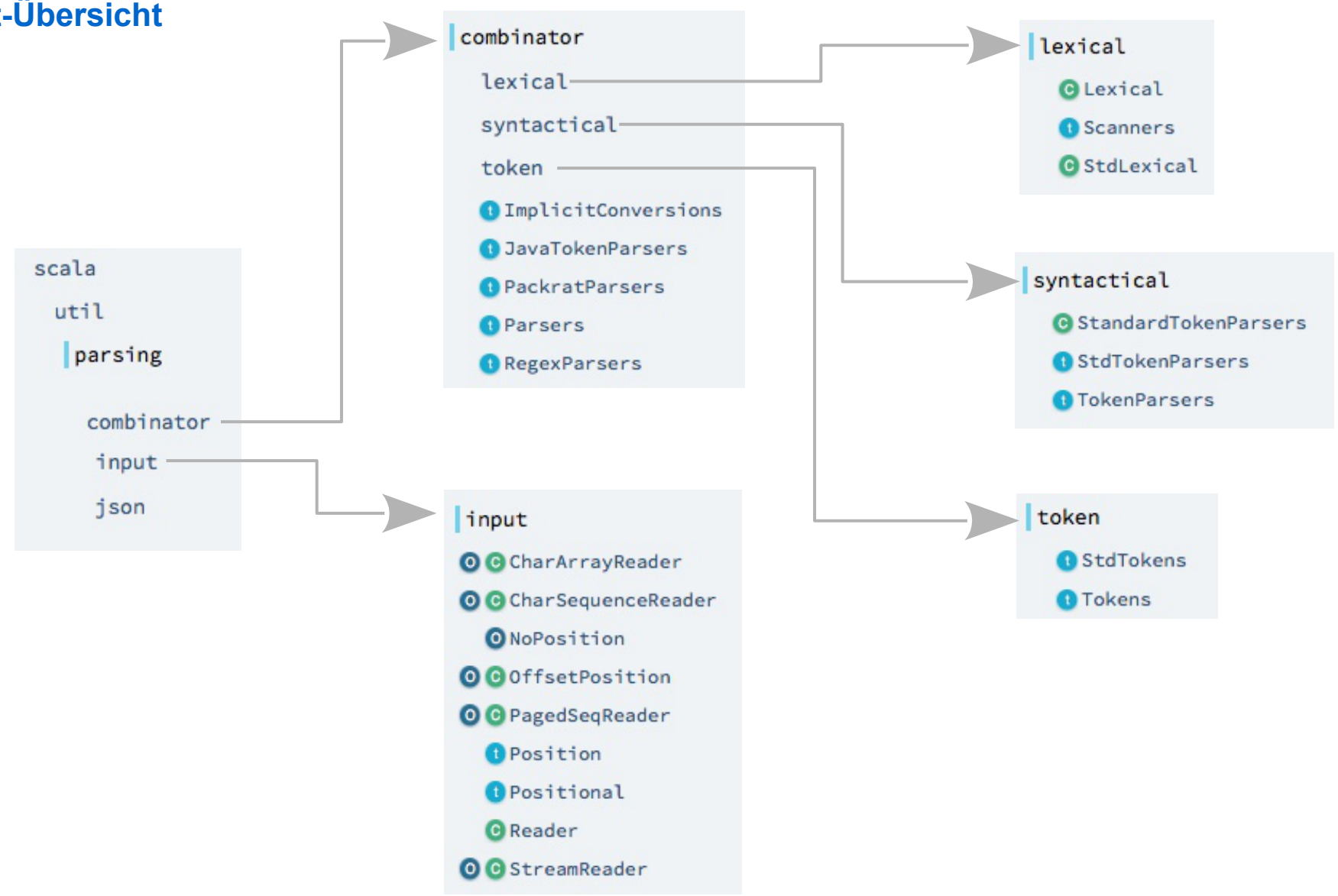
University of Applied Sciences

Parserkombinatoren: Ergänzungen

– Struktur der Combinator-API
– Beispiel Token-Parser mit Token-Scanner

Übersicht

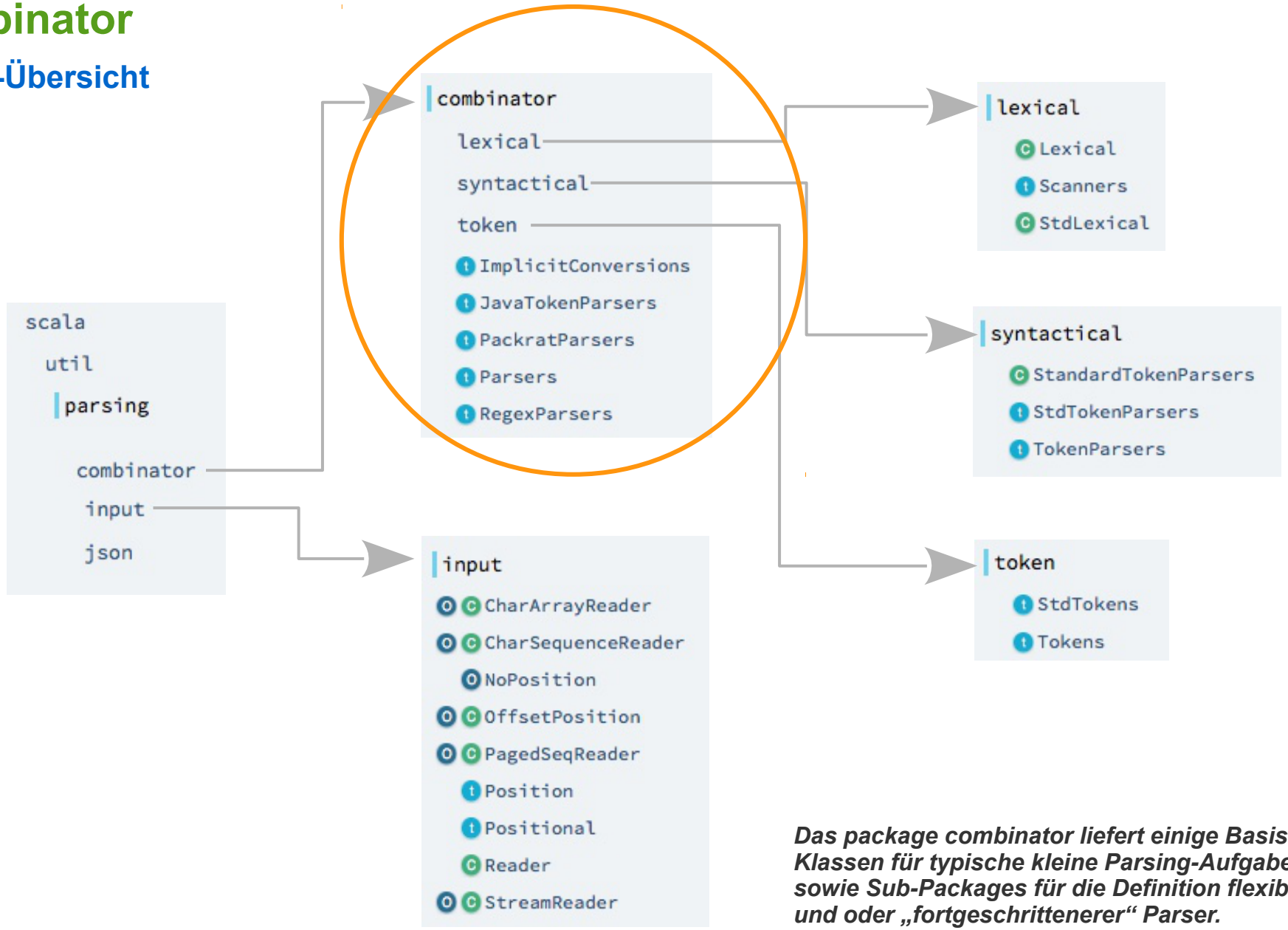
Paket-Übersicht



Package combinator

combinator

Paket-Übersicht



Das package combinator liefert einige Basis-Klassen für typische kleine Parsing-Aufgaben, sowie Sub-Packages für die Definition flexiblerer und oder „fortgeschrittenerer“ Parser.

Package Combinator

combinator

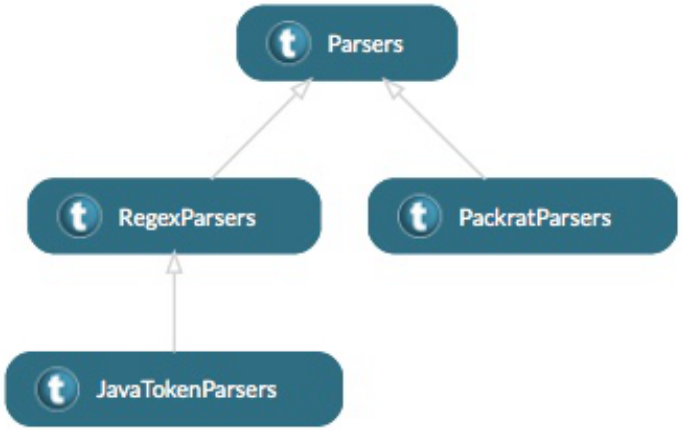
Zentrale Basis aller Parser

- **Parsers** – Die Basis aller Parser

Basis-Klassen für Anwendungsentwickler,
die mal eben schnell einen Parser benötigen (siehe Foliensatz 9):

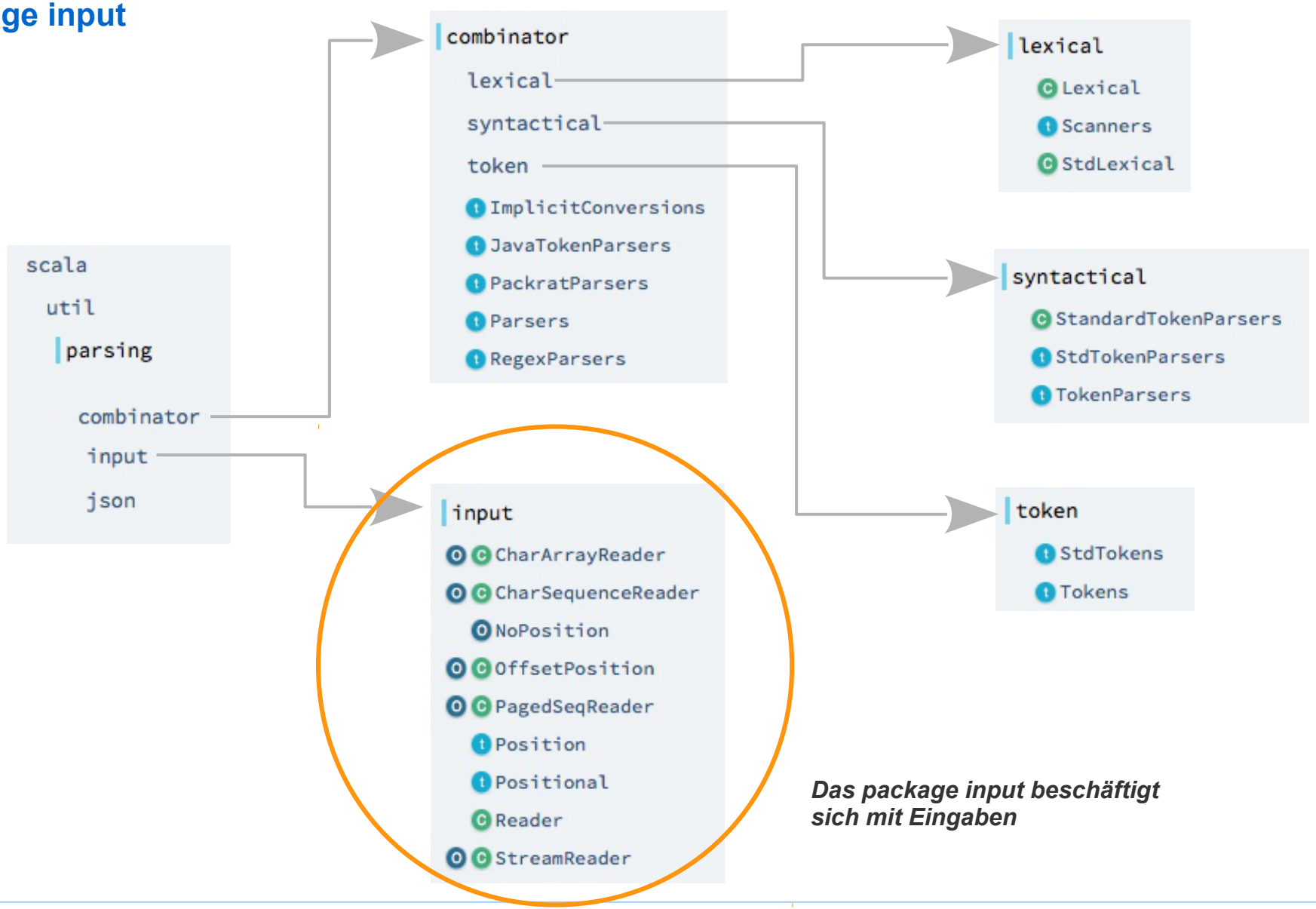
- **RegexParsers** – Parser als Regexe definieren
- **JavaTokenParsers** – Regex-Parser mit definierten „üblichen Tokens“ als RegExe
- **PackratParsers** – Füe Grammatiken mit Linksrekursion

```
combinator
  lexical
  syntactical
  token
  ImplicitConversions
  JavaTokenParsers
  PackratParsers
  Parsers
  RegexParsers
```



Übersicht

Package input



Das package input beschäftigt sich mit Eingaben

Package Input

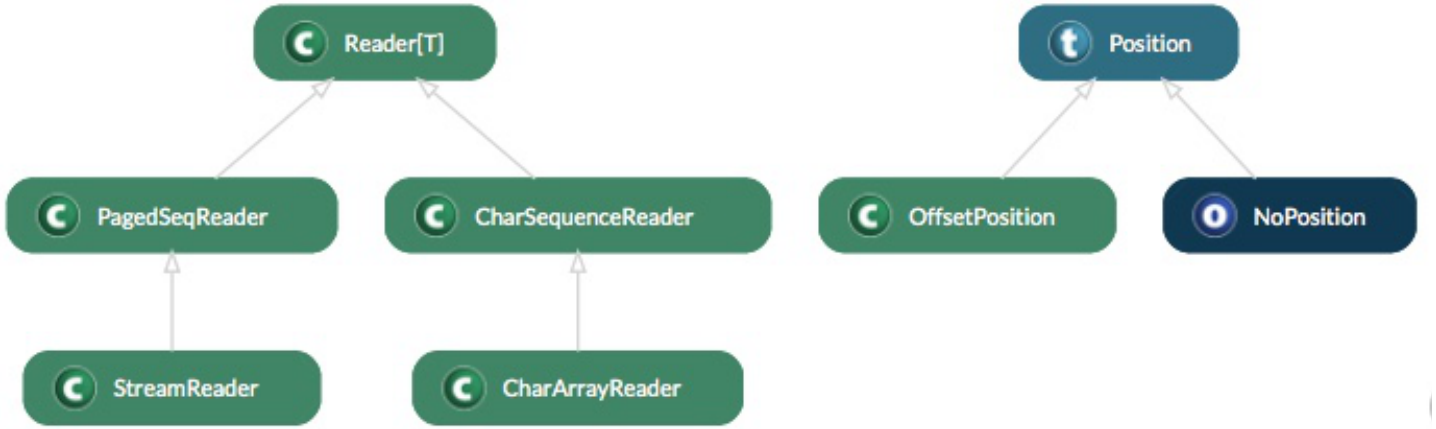
Input

Reader und Position

Wesentlicher Inhalt des Package sind die Traits / abstrakte Klassen:

- **Reader** – ein Strom von Werten die eine Position haben
- **Position** – eine Position und
  Positional – etwas mit einer Position

input
- CharArrayReader
- CharSequenceReader
- NoPosition
- OffsetPosition
- PagedSeqReader
- Position
- Positional
- Reader
- StreamReader



Es geht hier um die Verwaltung und erste Bearbeitung der Eingabe.
Also um das, was ein Scanner erledigt.

Die erhöhte Komplexität gegenüber den bisher betrachteten Scannern kommt daher, dass Parser-Kombinatoren Backtracking-Parser sind.

D.h. die Eingabe muss eventuell immer wieder zurück gesetzt werden.

Package Input

Reader

Reader – ein Strom von Werten die eine Position haben

- **PagedSeqReader extends Reader[Char]**

  A character reader reads a stream of characters (keeping track of their positions) from a

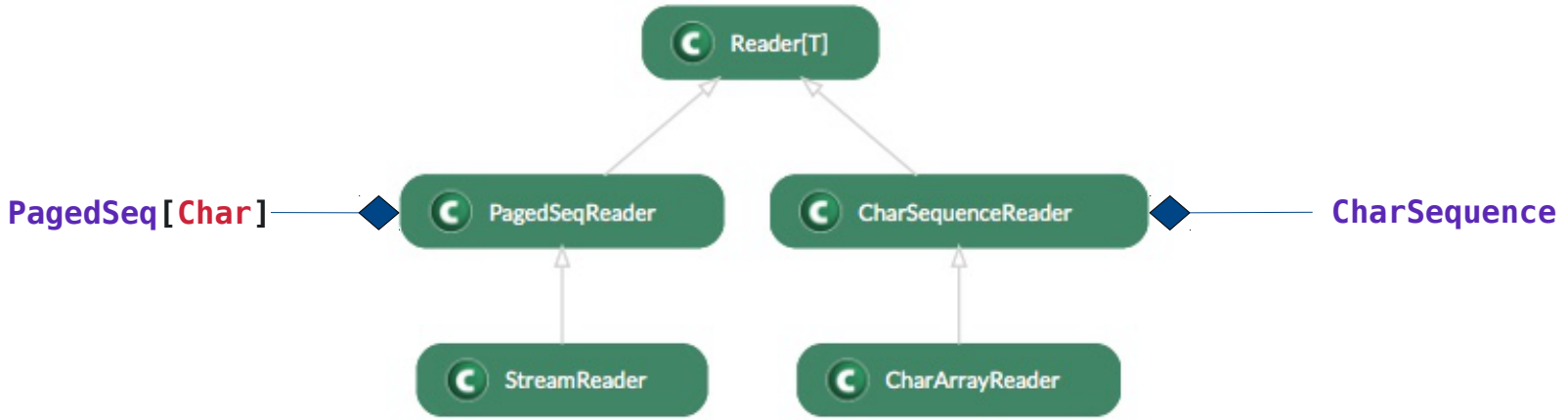
  `scala.collection.immutable.PagedSeq[Char]`

  PagedSeq*: An implementation of lazily computed sequences, where elements are stored in "pages", i.e. arrays of fixed size.)*

- **CharSequenceReader extends Reader[Char]**

  A character reader reads a stream of characters (keeping track of their positions) from an

  `java.lang.CharSequence`

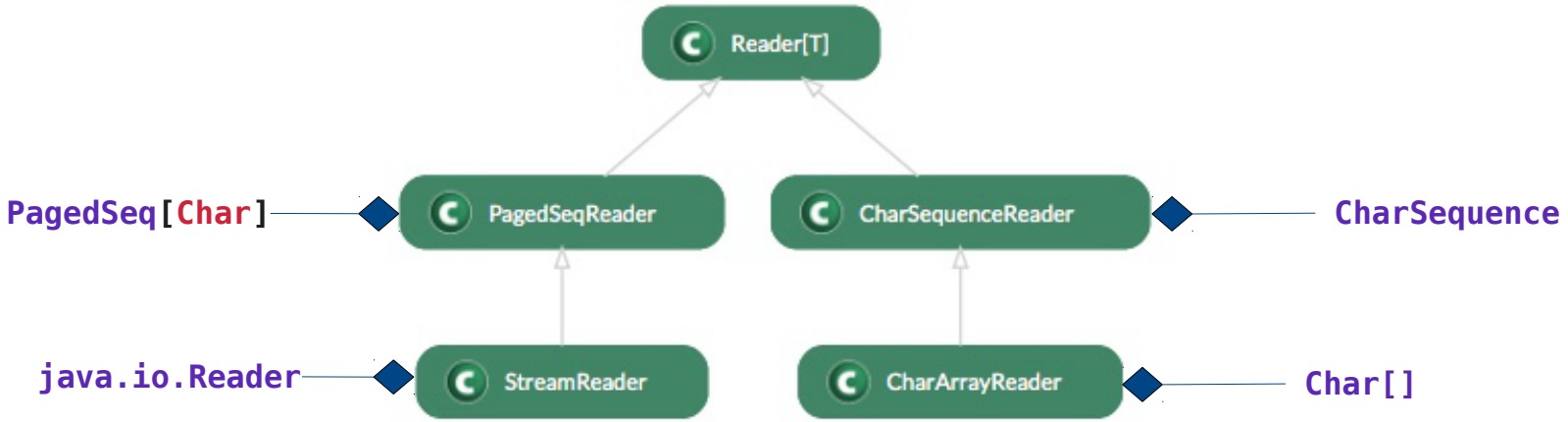


Die API-Doku scheint hier ein paar Nachlässigkeiten zu enthalten.

Reader

Reader – ein Strom von Werten die eine Position haben

- **StreamReader** extends PagedSeqReader
  Ein PagedReader der zeilenweise von einem java.io.Reader liest
- **CharArrayReader** extends CharSequencereader
  Ein CharSequenceReader der einem Array liest



Package Input

Reader

Basis-Abstraktion für Eingaben

vergleichbar mit java.io.Reader

```
import java.io.FileReader
import java.io.Reader

val reader: Reader = new FileReader("/some/path/to/aFile.txt")

var data: Int = reader.read
while ( data != -1 ) {
  val dataChar = data.toChar
  data = reader.read
  println(data) // prints a byte as Int value
}
```

Beispiel zu java.io.Reader

```
import java.io.FileReader
import util.parsing.input.StreamReader


var reader: StreamReader = StreamReader(new FileReader("/some/path/to/aFile.txt"))

while ( ! reader.atEnd ) {
  val dataChar = reader.first
  println(s"$dataChar at position ${reader.pos}") // prints Char and its position
  reader = reader.rest
}
```

Beispiel zu util.parsing.input.Reader

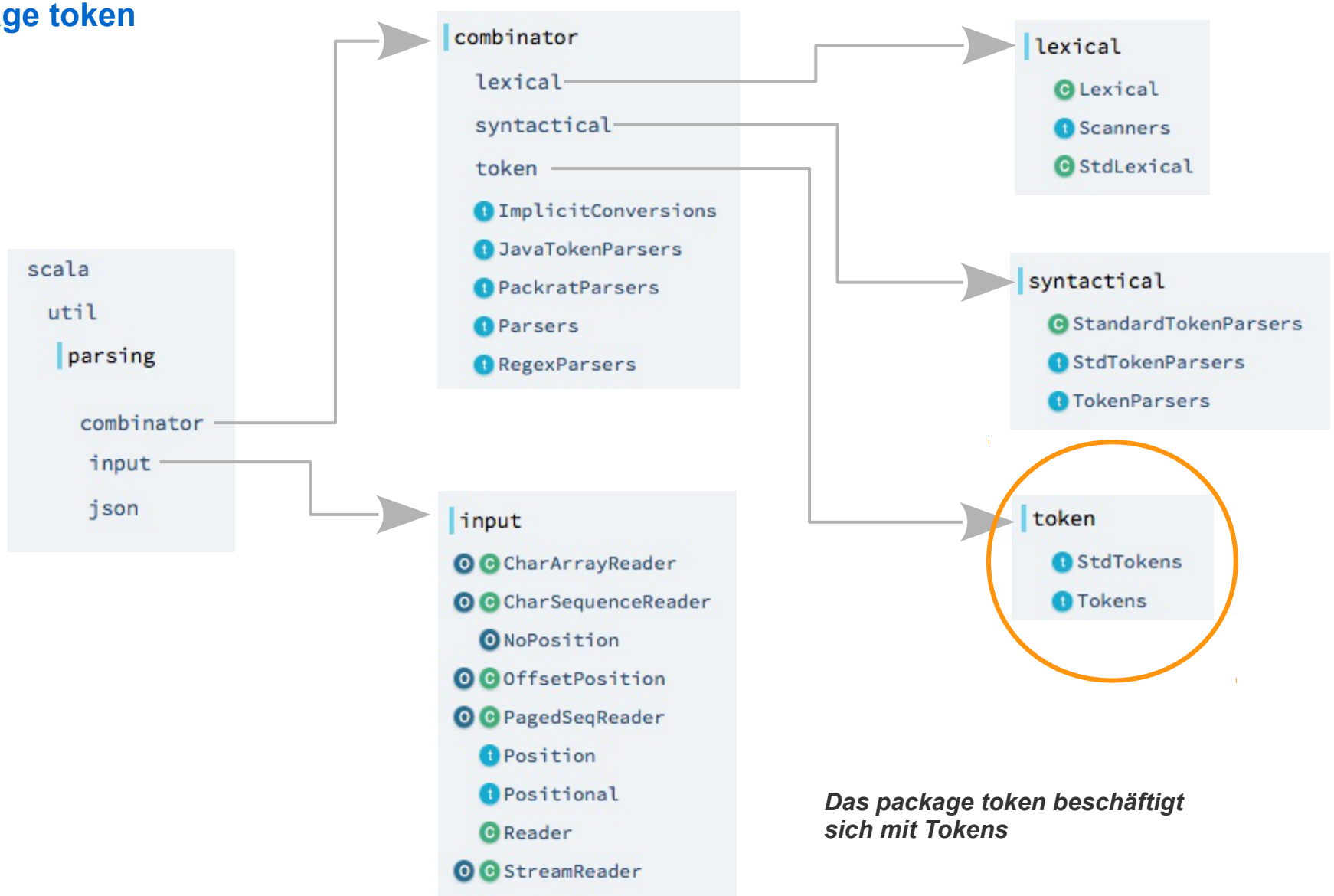
0123
abcd

aFile.txt

```
0 at position 1.1
1 at position 1.2
2 at position 1.3
3 at position 1.4

 at position 1.5
a at position 2.1
b at position 2.2
c at position 2.3
d at position 2.4

 at position 2.5
```

Übersicht

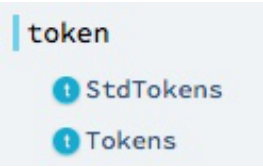
Package token



Das package token beschäftigt sich mit Tokens

Package token

token



Tokens

Basis für Token-Definitionen

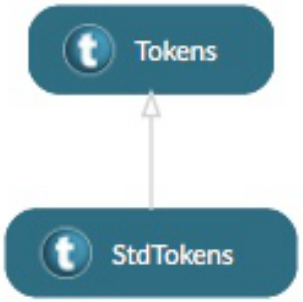
Definiert die Tokens `EOF` und `ErrorToken`

Eigene Token-Definitionen sollten in abgeleitete Klassen platziert werden

StdTokens

Vordefinierte Tokens für „Scala-artige“ Sprachen

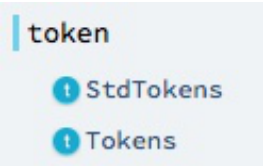
Kann als Beispiel für eine von Tokens abgeleitete Klasse verwendet werden.



Package token

Tokens

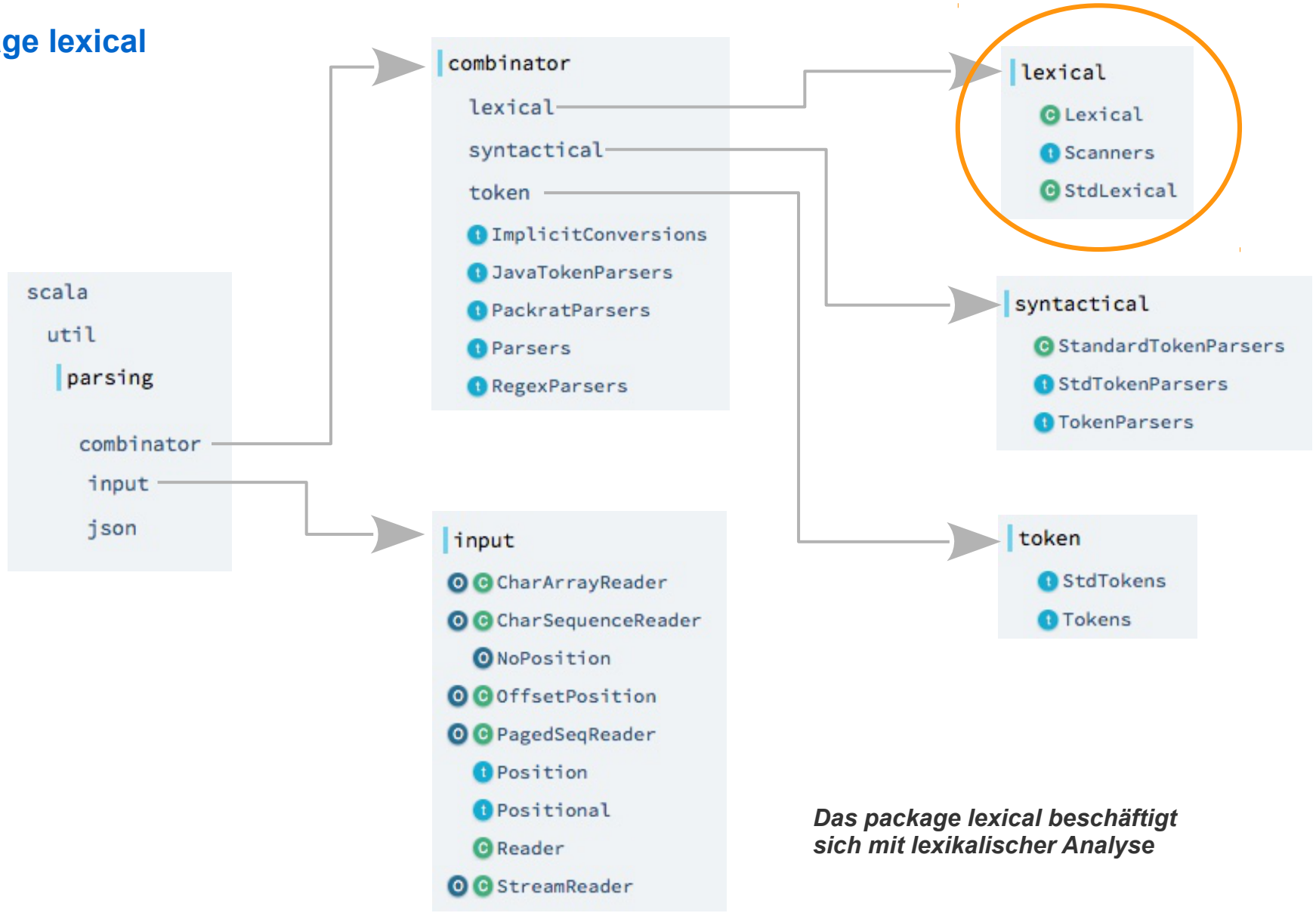
einfaches Beispiel für eigne Token-Definitionen




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

class ExpTokens extends Tokens {
  case class  NumberToken(chars: String)  extends Token
  case class  LeftPToken(chars: String)   extends Token
  case class  RightPToken(chars: String)  extends Token
  case class  AddOpToken(chars: String)   extends Token
  case class  MultOpToken(chars: String)  extends Token
}
```

Übersicht

Package lexical



Das package lexical beschäftigt sich mit lexikalischer Analyse

Package lexical

lexical

lexical
- Lexical
- Scanners
- StdLexical

Scanners

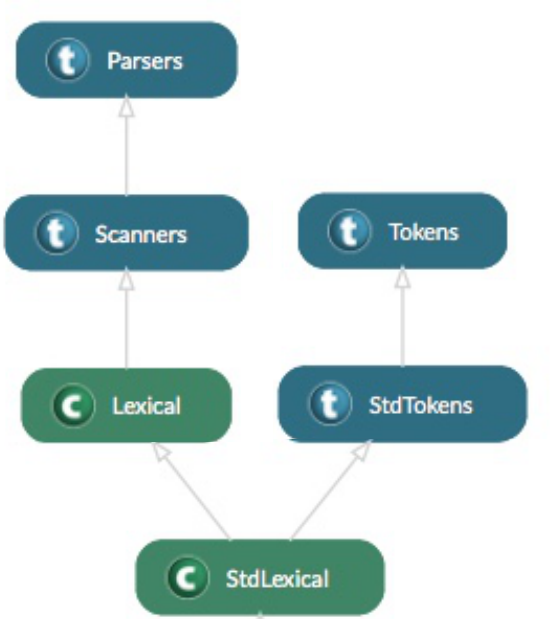
Trait mit der Kern-Funktionalität eines Scanners

Lexical

abstrakte Klasse liefert weitere Scanner-Funktionalität

StdLexical

Lexical Parser für eine Scala-artige Sprache (die typischen Scala- / Java-) Tokens sind definiert)



Package lexical

StdLexical

lexical
- Lexical
- Scanners
- StdLexical

Lexer für Scala-artige Sprachen

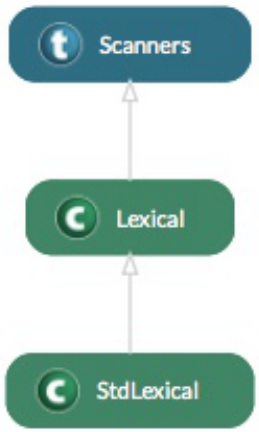
StdLexical definiert einen Parser für Tokens

```
def token: Parser[Token] = ...
```

Für einfache Sprachen mit kleinen Programmen eine gute handliche Lösung.

Für Sprachen mit größeren Programmen nicht akzeptabel da viel zu langsam: Tokens werden mit Hilfe eines Backtracking-Parsers erkannt!

Parsing-Techniken „dürfen“ natürlich zur Erkennung von Tokens eingesetzt werden. In einem „richtigen Compiler“ wird man dies aber unter allen Umständen vermeiden wollen.



„This component provides a standard lexical parser for a simple, Scala-like language. It parses keywords and identifiers, numeric literals (integers), strings, and delimiters.“ (API-Doku)

Package lexical

Scanners

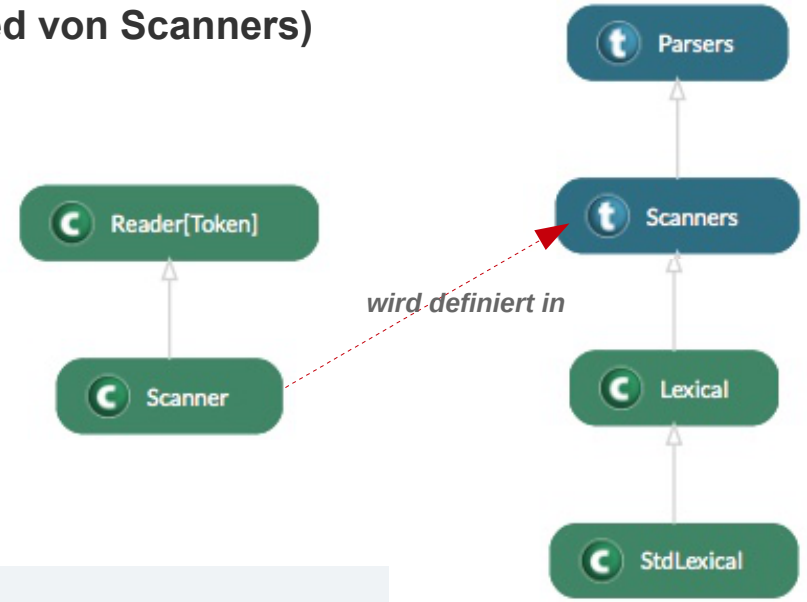
lexical
- Lexical
- Scanners
- StdLexical

Trait mit der Basisfunktionalität eines Scanners

Klasse Scanner wird in Scanners definiert

Ein Scanner ist ein Token-Reader

(Token ist abstraktes Typ-Mitglied von Scanners)



scala.util.parsing.combinator.lexical.Scanners

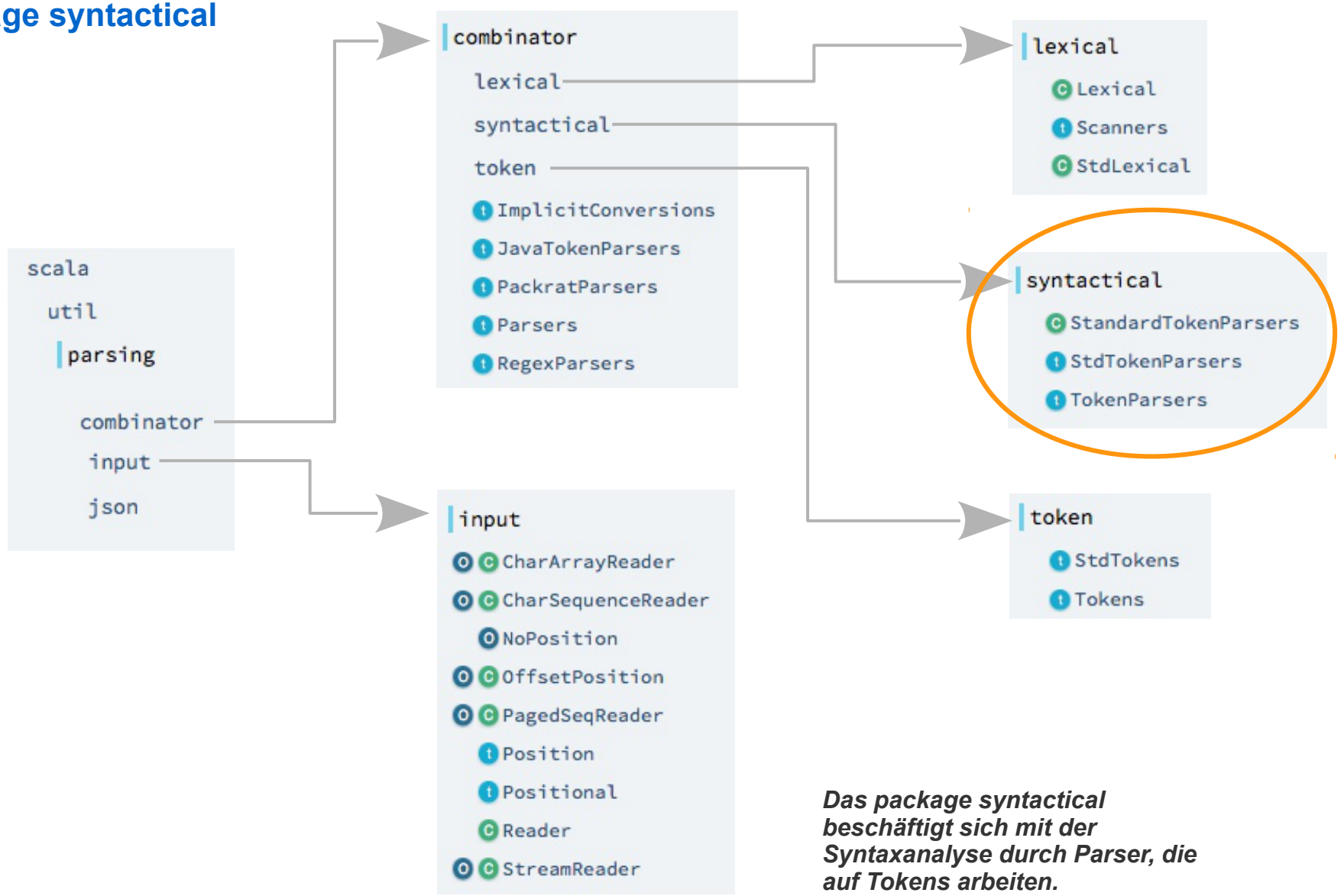
Scanner

```
class Scanner extends Reader[Token]
```

Scanner is essentially[1] a parser that produces **Token**s from a stream of characters. The tokens it produces are typically passed to parsers in **TokenParsers**.

Übersicht

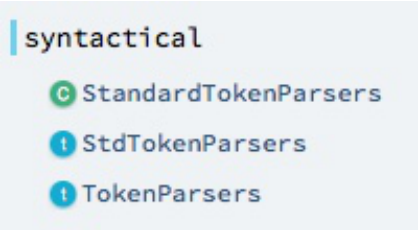
Package syntactical



Das package syntactical beschäftigt sich mit der Syntaxanalyse durch Parser, die auf Tokens arbeiten.

Package syntactical

syntactical



TokenParsers

Trait mit der Kern-Funktionalität eines Token-basierten Parsers

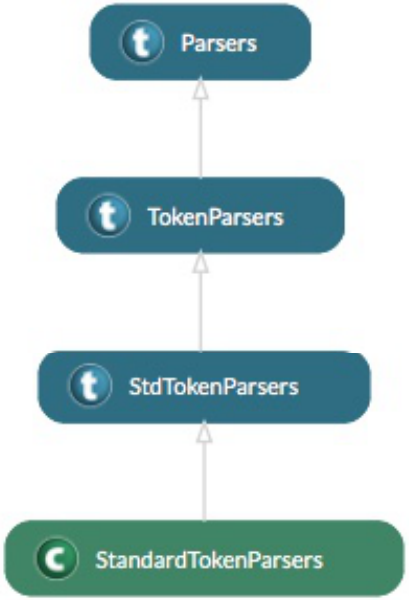
StdTokenParsers

liefert Parser für die Tokens in StdTokens

StandardTokenParsers

liefert Parser für die Tokens in StdTokens

marginale Erweiterung von StdTokenParsers

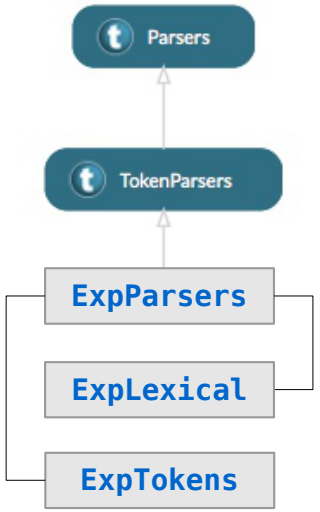


Beispiel: Token-Parser kooperiert mit eigenem Scanner

ExpParsers

TokenParser für arithmetische Ausdrücke auf Basis von TokenParsers

```
import scala.util.parsing.combinator.syntactical.TokenParsers
import scala.util.parsing.input.Reader

object ExpParsers extends TokenParsers {

  type Tokens = ExpTokens

  override val lexical : ExpLexical = new ExpLexical

  import lexical._

  val num_any : Parser[Any] =
    elem("number", _.isInstanceOf[NumberToken])

  def exp: Parser[Any]    = term ~ rep (addOp ~ term)
  def term: Parser[Any]   = factor ~ rep (multOp ~ factor)
  def factor: Parser[Any] = number | LeftPToken("(") ~ exp ~ RightPToken(")")

  def addOp: Parser[Any] = AddOpToken("+") | AddOpToken("-")
  def multOp: Parser[Any] = MultOpToken("*") | MultOpToken("/")
  def number: Parser[Any] = num_any

  def parse(str: String)  = {
    val lexer: Reader[lexical.Token] = new lexical.Scanner(str)

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




```
class ExpLexical extends ExpTokens {

  class Scanner(input: String) extends Reader[Token] {
    override def pos: Position = ???
    override def first: Token = ???
    override def rest: Reader[Token] = ???
    override def atEnd: Boolean = ???
  }
}
```

Beispiel: Token-Parser kooperiert mit eigenem Scanner

ExpParsers TokenParser für arithmetische Ausdrücke auf Basis von TokenParsers

```
import scala.util.parsing.combinator.syntactical.TokenParsers
import scala.util.parsing.input.Reader

object ExpParsers extends TokenParsers {

  type Tokens = ExpTokens

  override val lexical : ExpLexical = new ExpLexical

  import lexical._

  val num_any : Parser[Any] =
    elem("number", _.isInstanceOf[NumberToken])

  def exp: Parser[Any]    = term ~ rep (addOp ~ term)
  def term: Parser[Any]   = factor ~ rep (multOp ~ factor)
  def factor: Parser[Any] = number | LeftPToken("(") ~ exp ~ RightPToken(")")

  def addOp: Parser[Any] = AddOpToken("+") | AddOpToken("-")
  def multOp: Parser[Any] = MultOpToken("*") | MultOpToken("/")
  def number: Parser[Any] = num_any

  def parse(str: String)  = {
    val lexer: Reader[lexical.Token] = new lexical.Scanner(str)
    phrase(exp)(lexer) match {
      case Success(tree,_)       => tree
      case error@NoSuccess(_,_)  => println(error)
        throw new IllegalArgumentException("Parser Error")
    }
  }
}
```

TokenParsers hat eine implizite Konversion von Tokens zu Parsern dieser Tokens.

Diese wird hier bei der der addOp-, multOp- und number-Kombinatoren verwendet.

Die Methode elem ist in Parsers definiert.

num_any wird definiert, da wir nicht ein ganz bestimmtes NumberToken erkennen wollen, sondern irgendeins.

Beispiel: Token-Parser kooperiert mit eigenem Scanner

ExpParsers TokenParser für arithmetische Ausdrücke auf Basis von TokenParsers

```
import scala.util.parsing.combinator.syntactical.TokenParsers
import scala.util.parsing.input.Reader

object ExpParsers extends TokenParsers {
  import AST._
  type Tokens = ExpTokens
  override val lexical : ExpLexical = new ExpLexical
  import lexical._

  val num_any : Parser[Any] = elem("number", _.isInstanceOf[NumberToken])

  def number: Parser[Number] = num_any   ^^ { case (x: Any) => Number(x.asInstanceOf[NumberToken].chars.toInt)}

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

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

  def term = chainl1(factor, factor, multOp)

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

  def factor: Parser[Exp] =
    number   |
    LeftPToken("(") ~> exp <~ RightPToken(")")

  def parse(str: String)  = {
    val lexer: Reader[lexical.Token] = new lexical.Scanner(str)
    phrase(exp)(lexer) match {
      case Success(tree,_)       => tree
      case error@NoSuccess(_,_)  => println(error)
        throw new IllegalArgumentException("Parser Error")
    }
  }
}
```

Der chainl1-Kombinator und die „Baum-Transformatoren“ ^^ und ^^^ funktionieren natürlich auch mit dem Token-Parser mt eigenen Scanner.

Anpassung des Scanners

Der Scanner muss jetzt noch in den Kontext der Combinator-API angepasst werden:

Er muss

- **als Token-Reader (also mit funktionalem Verhalten) re-definiert und**
- **und in eine lexikalische Komponenten**

eingebettet werden.

```
class ExpLexical extends ExpTokens {

  ...

  class Scanner(input:   String,
     private val actPos: Int = 0) extends Reader[Token] {

     ...

     override def pos: Position = ???

     override def first: Token = ???

     override def rest: Reader[Token] = ???

     override def atEnd: Boolean = ???

  }

}
```

Hier kann jetzt eine angepasste Version des

des RegEx-basierten Scanners eingesetzt werden.

Statt eines veränderlichen Zustands liefert dieser Scanner, mit* rest*, eine neue Instanz. Da der Zustand ja nur aus der aktuellen Position besteht, müssen sich die Scanner-Instanzen nur durch diese Position unterscheiden. Damit sind effiziente Implementierungen möglich.