Compilerbau CS1019

Th. Letschert

TH Mittelhessen Gießen

University of Applied Sciences

Lexikalische Analyse: Scanner

– Scanner

– Scanner-Implementierung

Lexikalische Elemente / Tokens

Beispiel: Syntax der „Programmiersprache“ der vollständig geklammerten Ausdrücke

Expression → *Number* | *Operation*
Number → 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 0
Operation → (*Expression Operator Expression*)
Operator → + | – | * | /
Startsymbol: *Expression*
Nichtterminale: *Expression, Number Operator*
Terminale: 1 2 3 4 5 6 7 8 9 0 + – * /

Diese Syntax hat zwei Nachteile

- Alle Ausdrücke müssen vollständig geklammert sein
- Es sind nur Ziffern erlaubt
  Um „richtige Zahlen“ zu integrieren, muss die Syntax nur ein wenig erweitert werden.

Token

Beispiel: Erweiterung der Ausdruckssprache um Zahlen

Um „richtige Zahlen“ zu integrieren, muss die Syntax nur ein wenig erweitert werden (ε ist der leere Text):

Expression → *Number* | *Operation*
Operation → (*Expression Operator Expression*)
Operator → + | − | * | /
Number → *NonNullDigit DigitList*
DigitList → *Digit DigitList* | ε
NonNullDigit → 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
Digit → 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

Die Definition des Compilers aus Foliensatz 1 entsprechend zu erweitern ist kein besonderes Problem

Syntaxbäume für Ausdrücke mit richtigen Zahlen sind schnell definiert:

```
sealed abstract class Exp
case class NumberExp(n: Number) extends Exp
case class OperationExp(exp1: Exp, op: Operator, exp2: Exp) extends Exp
case class Operator(opSymbol: Char)

case class Number(nnDigit: Char, digitList: DigitList)
sealed abstract class DigitList
case object EmptyDigitList extends DigitList
case class NonEmptyDigitList(digit: Char, rest: DigitList) extends DigitList
```

Token: Erkennung im Parser

Beispiel: Erweiterung der Ausdruckssprache um Zahlen

Der Parser ist ebenfalls leicht zu erweitern:

```
def parse(text: String): Exp = {
  var pos: Int = 0

  def parseExp: Exp = text(pos) match {
    case '1' => NumberExp(parseNumber)
    …
    case '9' => NumberExp(parseNumber)
    case '(' => parseOperation
    case _ => throw new IllegalArgumentException
  }
  def parseNumber: Number = text(pos) match {
    case '1' => { pos = pos+1; Number('1', parseDigitList) }
    …
    case '9' => { pos = pos+1; Number('9', parseDigitList) }
    case _ => throw new IllegalArgumentException
  }

  def parseDigitList : DigitList =
    if ((pos >= text.length()) || !(text(pos) == '0' || text(pos) == '1' || text(pos) == '2' ||
          text(pos) == '3' || text(pos) == '4' || text(pos) == '5' || text(pos) == '6' ||
          text(pos) == '7' || text(pos) == '8' || text(pos) == '9' || text(pos) == '0') ) EmptyDigitList
    else {
      pos = pos+1;
      NonEmptyDigitList(text(pos-1), parseDigitList)
    }
    def parseOperation: OperationExp = { Unverändert }
    parseExp
}
```

Token: Lexikalische Ebne und Syntaktische Ebnene

Beispiel: Erweiterung der Ausdruckssprache um Zahlen

Und auch der Codegenerator sollte keine Probleme machen. Die Ziffernfolgen müssen einfach nur zu Zahlen „verrechnet“ werden.

Das Ganze hätten wir auch einfacher bekommen können:

- Die Syntax der Zahlen kann durch einen regulären Ausdruck beschrieben werden,
- Für reguläre Ausdrücke gibt es einen allgemein verfügbaren, bewährten, effizienten Erkennungsmechanismus

In Kombination:

Expression → ***Number*** | *Operation*
Operation → (*Expression Operator Expression*)

} Grammatik / Syntaktische Ebene

Operator → + | − | * | /
Number = [1-9][0-9]*

} Reg.-Ausdr. / Lexikalische Ebene

} Syntax

Die beiden Ebenen müssen kombiniert werden.

Lexikalische Elemente / Tokens

Aufteilung der Syntax in zwei Ebenen

Traditionell teilt man die Syntax einer Sprache in zwei Ebenen auf.

- Eine „untere“ Ebene der „Worte“, die mit regulären Ausdrücken beschrieben und erkannt werden können und
- eine höhere der „Sätze“, die mit Grammatiken beschrieben wird und für die mächtigere Mechanismen der Syntaxanalyse eingesetzt werden.

Diese Unterteilung ist nicht zwingend notwendig aber oft hilfreich:

- Reguläre Ausdrücke eignen sich nicht für die Definition und das Erkennen rekursiver Strukturen,
- Sie sind aber bequem in der Anwendung und haben sehr effiziente Implementierungen
- Die Syntaxanalyse kann mit der Aufteilung in zwei Ebenen vereinfacht und beschleunigt werden

Lexikalische Analyse

Die Syntaxanalyse der unteren Ebene, die „Worte“ erkennt, wird lexikalische Analyse genannt.

Lexikalisches Element / Token

Die vollständigen Strukturen auf unterster Ebene, die „Worte“, werden Tokens oder lexikalische Elemente genannt. Aus ihnen werden dann die „Sätze“ (Programme / Übersetzungseinheiten) gebildet.

Syntax: Lexikalische und Syntaktische Ebene

Beispiel Aufteilung der Syntax der Ausdruckssprache in zwei Ebenen

Syntaxtische Ebene

Exp → *NumberExp* | *OperationExp*
NumberExp → NumberToken
OperationExp → LeftPToken *Expression* OperatorToken *Expression* RightPToken

Lexikalische Ebene

NumberToken = [1..9][0..9]*
OperatorToken = + | – | * | /
LeftPToken = (
RightPToken =)

Token

Definition

- Ein Token ist eine Teilsequenz der Gesamtsequenz der Zeichnen
- Tokens sind
  - Meist durch weise Zeichen separiert
  - Können aber auch unmittelbar aufeinander folgen
- Ein weises Zeichen ist ein Zeichen im Eingabetext das ohne inhaltliche Relevanz ist.
  Z.B. Leerzeichen, Tabs, Zeilenumbruch, etc.

Token-Definition und Token-Erkennung

- Die Tokens sind meist Elemente einer regulären Sprache,
- Sie werden dann mit regulären Ausdrücken beschrieben und mit endlichen Automaten erkannt
- Es gibt keinen zwingenden Zusammenhang zwischen
  - der lexikalischen Ebene (Erkennen von Worten oder Sätzen) und
  - dem Verfahren zur Beschreibung Sprache oder der Analyse von Texten auf dieser Ebene
- Wir gehen aber i.d.R. davon aus, dass Tokens Elemente einer regulären Sprache sind

Token

Arten von Tokens

In Programmiersprachen gibt es üblicherweise eine begrenzte Menge an unterschiedlichen Tokens:

- **Schlüsselwort / *Keyword***

  Ein Wort das der Strukturierung des Programmtextes dient und für sich allein keine Bedeutung hat.

  Beispiele: begin / end / def / class etc.

- **Bezeichner / *Identifier***

  Ein Name für etwas das im Programm definiert wurde (Variable, Typ, Funktion, …)

  Ein Bezeichner hat in unterschiedlichen Programmen und meist auch an unterschiedlichen Stellen in einem Programme eine unterschiedliche Bedeutung.

  Bezeichner haben eine Glültigkeitsbereich (*Scope*) – das ist der Bereich innerhalb eines Programms, an dem ihre Bedeutung fixiert ist.

  Eine Variablendefinition legt beispielsweise einen Bezeichner als Name der Variablen fest, diese Festlegung gilt aber nur für einen bestimmten Bereich des Programms.

- **Literal / *Literal***

  Ein Literal bezeichnet einen Wert, z.B. eine Zahl, ein Zeichen, einen String, etc.

  Das gleiche Literal hat immer in allen Programmen und an jeder Stelle in einem Programm die gleiche Bedeutung.

- **Begrenzer / Delimiter**

  Einzelne Zeichner die zur Strukturierung des Programmtextes genutzt werden, wie etwa die unterschiedlichen Klammerzeichen, Komma, Semikolon, etc.

API-Unterstützung

java.util.Scanner

Die Klasse Scanner kann genutzt werden um einen Scanner zu implementieren.

Allerdings: *java.util.Scanner* ist nicht gedacht als „Scanner“ im hier gemeinten Sinn.

Es dient als Unterstützung für die einfache Analyse von Benutzereingaben – vergleichbar mit *scanf* in C.

scala.util.Matching

Reguläre Ausdrücke werden von der API unterstützt und sind eine gute Hilfe bei der Implementierung eines Scanners.

Hinweise: Studieren Sie die API-Dokumentation von scala.util.matching.Regex (http://www.scala-lang.org/api/2.12.4/scala/util/matching/Regex.htm)

Scanner – auch *Lexer* oder *Tokenizer* genannt

Aufgabe Scanner

Scanner: SW-Komponenten die Text in eine Folge von Tokens transformiert.

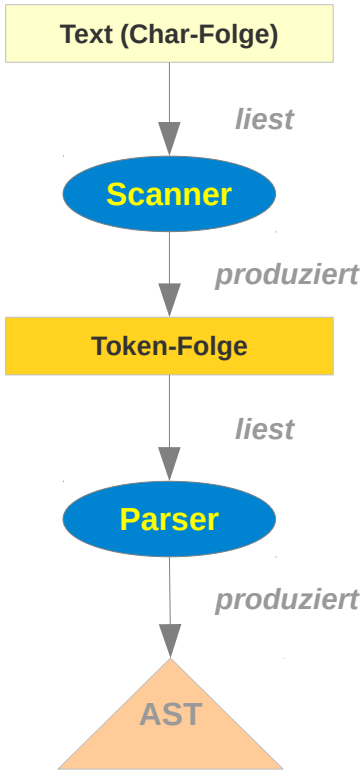
Implementierung

- **Komplett „händisch“**
  Regex => NEA => DEA => DEA-Implementierung
- **Händisch mit Hilfe einer Regex-Implementierung**
- **Mit Hilfe eines Scannergenerators**
- **...**

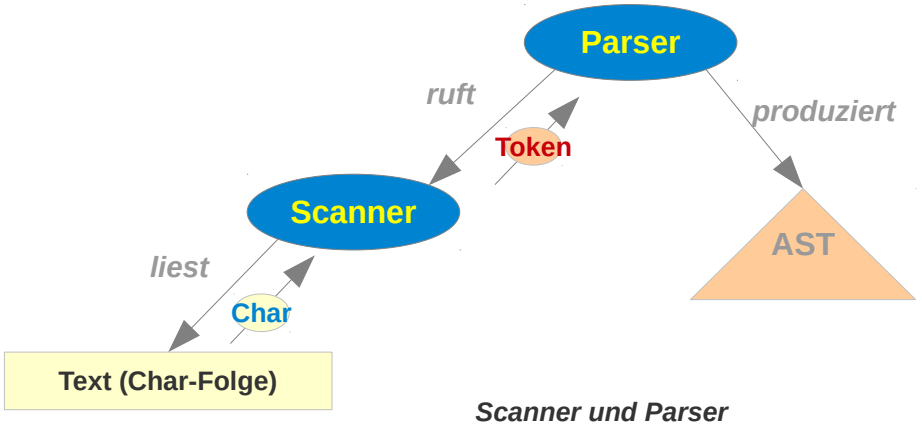
Interaktion Scanner – Parser

Der Scanner kann auf unterschiedliche Arten implementiert werden:

- **2-Phasen: Die einfachste ist eine Funktion: Text => Token-Folge**
- **Verschränkt: Üblich ist eine Funktion, die vom Parser aufgerufen wird und das jeweils nächste Token liefert.**



Scanner und Parser operieren in 2 Phasen



Scanner und Parser operieren verschränkt

Scanner-Implementierung

Scanner-Beispiel 1: Scanner für die Ausdruckssprache mit Zahlen

Implementierung mit RegEx-Funktionen der Scala-API (Scanner und Parser verschränkt.)

```
class ExpScanner(input: String) {

  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 tokenPatS = (List(numberPatS, operatorPatS, leftPPatS, rightPPatS).reduce( _ + "|" + _ ))
  private val tokenPat = tokenPatS.r

  import ExpScanner._

  val tokens = tokenPat.findAllIn(input)

  def nextToken: Token =
    if (tokens.hasNext)
      tokens.next() match {
        case NumberPat(num) =>  NumberToken(num.toInt)
        case OperatorPat(c) =>  OperatorToken(c.charAt(0))
        case LeftPPat(s) =>     LeftPToken
        case RightPPat(s) =>    RightPToken
       case x => throw new Exception("??: "+x)
      }
   else EOFToken

}
```

```
object ExpScanner {
  // Definition of Tokens
  sealed abstract class Token
  case class NumberToken(d: Int) extends Token
  case class OperatorToken(c: Char) extends Token
  case object LeftPToken extends Token
  case object RightPToken extends Token
  case object EOFToken extends Token
}
```

Scanner-Implementierung

Scanner-Beispiel 1: Scanner / Parser verschränkt

Parser

```
class ExpParser(scanner: ExpScanner) {

  import ExpScanner._

  def parse: ExpTree = scanner.nextToken match {
    case NumberToken(x) => Number(x)
    case LeftPToken => parseOperation
    case _ => throw new IllegalArgumentException
  }

  def parseOperation: Operation = {
    val exp1 = parse
    val opTok = scanner.nextToken
    opTok match {
      case OperatorToken(opSymbol) => {
        val exp2 = parse
        val rightpTok = scanner.nextToken
        rightpTok match {
          case RightPToken =>
          case _ => throw new IllegalArgumentException("Expected ), found " + rightpTok)
        }
        opSymbol match {
          case '+' => Operation(exp1, Operator(opSymbol), exp2)
          case '-' => Operation(exp1, Operator(opSymbol), exp2)
          case '*' => Operation(exp1, Operator(opSymbol), exp2)
          case '/' => Operation(exp1, Operator(opSymbol), exp2)
          case _ => throw new IllegalArgumentException("Expected operator char, found " + opSymbol)
        }
      }
      case _ => throw new IllegalArgumentException("Expected operator, found " + opTok)
    }
  }
}
```

Scanner-Implementierung

Scanner-Beispiel 2: Scanner für eine Zeilen-orientierte Sprachen

Zeilenorientierung: Das Zeilenende ist ein relevantes Token

Viele Sprachen sind zeilenorientiert (z.B. Scala, alle Assemblersprachen)

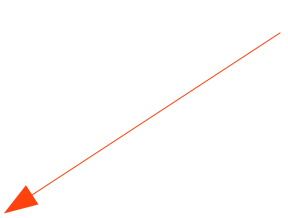
Umgang mit Zeilen:

- **jede Zeile wird für sich geparst**
  Nur möglich, wenn Zeilen syntaktische Strukturen auf oberster Ebene repräsentieren
- **Zeilenende wird als Token interpretiert**
  Ermöglichst syntaktische Strukturen die über Zeilengrenzen hinaus gehen.

Scanner-Implementierung

Scanner-Beispiel 2: Scanner für eine Zeilen-orientierte Sprachen Zeilenende ~> EOL-Token (1)

```
class CmdScanner(input: String) {

  private val numberPatS    = """(0|(?:[1-9][0-9]*))"""
  private val operatorPatS  = """(ADD|SUB|MUL|DIV|JMP)"""
  private val identPatS     = """(\w+)"""
  private val registerPatS  = """(?:\$([0-9][0-9]?))"""
  private val labelPatS     = """(?:(\w+)[ ]*:)"""
  private val newLinePatS   = """(\n)"""

  private val NumberPat   = numberPatS.r
  private val OperatorPat = operatorPatS.r
  private val identPat    = identPatS.r
  private val registerPat = registerPatS.r
  private val labelPat    = labelPatS.r
  private val newLinePat  = newLinePatS.r

  private val tokenPatS = List(numberPatS, operatorPatS, registerPatS, labelPatS, identPatS, newLinePatS).
            reduce( _ + "|" + _ )

  private val tokenPat = tokenPatS.r

  import CmdScanner._
  import CmdScanner.Operation._

  val tokens = tokenPat.findAllIn(input)

  def nextToken: Token =
    if (tokens.hasNext)
      tokens.next() match {
        case NumberPat(num)   => NumberToken(num.toInt)
        case OperatorPat(op)  => OperatorToken(op)
        case registerPat(reg) => RegisterToken(reg.toInt)
        case labelPat(l)      => LabelToken(l)
        case identPat(id)     => IdToken(id)
        case newLinePat(nl)   => EOLToken
        case x                => ErrorToken("unexpected "+x)
      }
    else EOFToken

}
```



Scanner-Beispiel 2: Scanner für eine Zeilen-orientierte Sprachen

Zeilenende ~> EOL-Token (2)

```
object CmdScanner {  // Definition of Tokens

  object Operation extends Enumeration {
    type Operation = Value
    val Add, Sub, Mult, Div, Jmp  = Value
    implicit def fromName(name: String): Value =
      values.find(_.toString.toLowerCase == name.toLowerCase()).
           getOrElse(throw new Exception("udefined Op"))
  }

  import Operation._

  sealed abstract class Token
  case class  OperatorToken(op: Operation)  extends Token
  case class  NumberToken(d: Int)           extends Token
  case class  RegisterToken(d: Int)         extends Token
  case class  LabelToken(name: String)      extends Token
  case class  IdToken(name: String)         extends Token
  case object EOLToken                      extends Token
  case object EOFToken                      extends Token
  case class  ErrorToken(msg: String)       extends Token
}
```

Scanner-Implementierung

Scanner-Beispiel 3: Scanner nutzt Regex.findPrefixMatchOf

Prefix Match: Match am Anfang eines Strings

Aktionen:

- **Whitespace am Anfang eliminieren**
- **Nach passendem Match suchen**
- **findPrefixMatch kann genutzt werden ein passendes Muster zu finden und dabei die Kontrolle über die Position zu behalten**

Scanner-Implementierung

Scanner-Beispiel 3: Scanner nutzt Regex.findPrefixMatchOf (1)

```
class CmdScanner(input: String) {

  …       Pattern Definitionen

  private var offset = 0
  private val source: CharSequence = input

  private val pats: List[Regex] = List(NumberPat, OperatorPat, registerPat, labelPat, identPat, newLinePat)

  private def skipWhiteSpace(): Int =
    whitespacePat.findPrefixMatchOf(input.subSequence(offset, input.length())) match {
        case Some(matched) => offset + matched.end
        case None => offset
      }

  private def matchRegex(r: Regex): Option[String] = {
    r.findPrefixMatchOf(source.subSequence(offset, input.length())) match {
      case Some(matchedMatch) =>
        val res = Some(source.subSequence(offset, offset + matchedMatch.end).toString)
        offset = offset + matchedMatch.end
        res
      case None => None
    }
  }
```

Reihenfolge ist wichtig!

Scanner-Implementierung

Scanner-Beispiel 3: Scanner nutzt Regex.findPrefixMatchOf (2)

```
class CmdScanner(input: String) {

  …
  def nextToken: Token = {
    offset = skipWhiteSpace()

    var matched: Option[String] = None
    val startOffset = offset
    var i = 0

    while (! matched.isDefined && i < pats.length) {
      matched = matchRegex(pats(i))
      if (!matched.isDefined) { offset = startOffset }
      i = i+1
    }

    matched match {
      case None =>
        if (offset >= input.lastIndexOf())
          EOFToken
        else ErrorToken("unexpected end of input")
      case Some(matchedStr) =>
        matchedStr match {
          case NumberPat(num)   => NumberToken(num.toInt)
          case OperatorPat(op)  => OperatorToken(op)
          case registerPat(reg) => RegisterToken(reg.toInt)
          case labelPat(l)      => LabelToken(l)
          case identPat(id)     => IdToken(id)
          case newLinePat(nl)   => EOLToken
          case x                => ErrorToken("unexpected "+x)
        }
    }
  }

}
```

Die Position des Tokens kann (und sollte) in diesem gespeichert werden, damit der Parser genauere Fehlermeldungen generieren kann.

Bonusmaterial

Scanner-Implementierung mit Parser-Kombinatoren

Parser-Kombinatoren

Scala umfasst eine kleine Bibliothek zur Unterstützung der Syntaxanalyse

Diese kann natürlich auch eingesetzt werden um einen Scanner zu implementieren

SBT-Konfiguration:

```
libraryDependencies += "org.scala-lang.modules" %% "scala-parser-combinators" % "1.0.6"
```

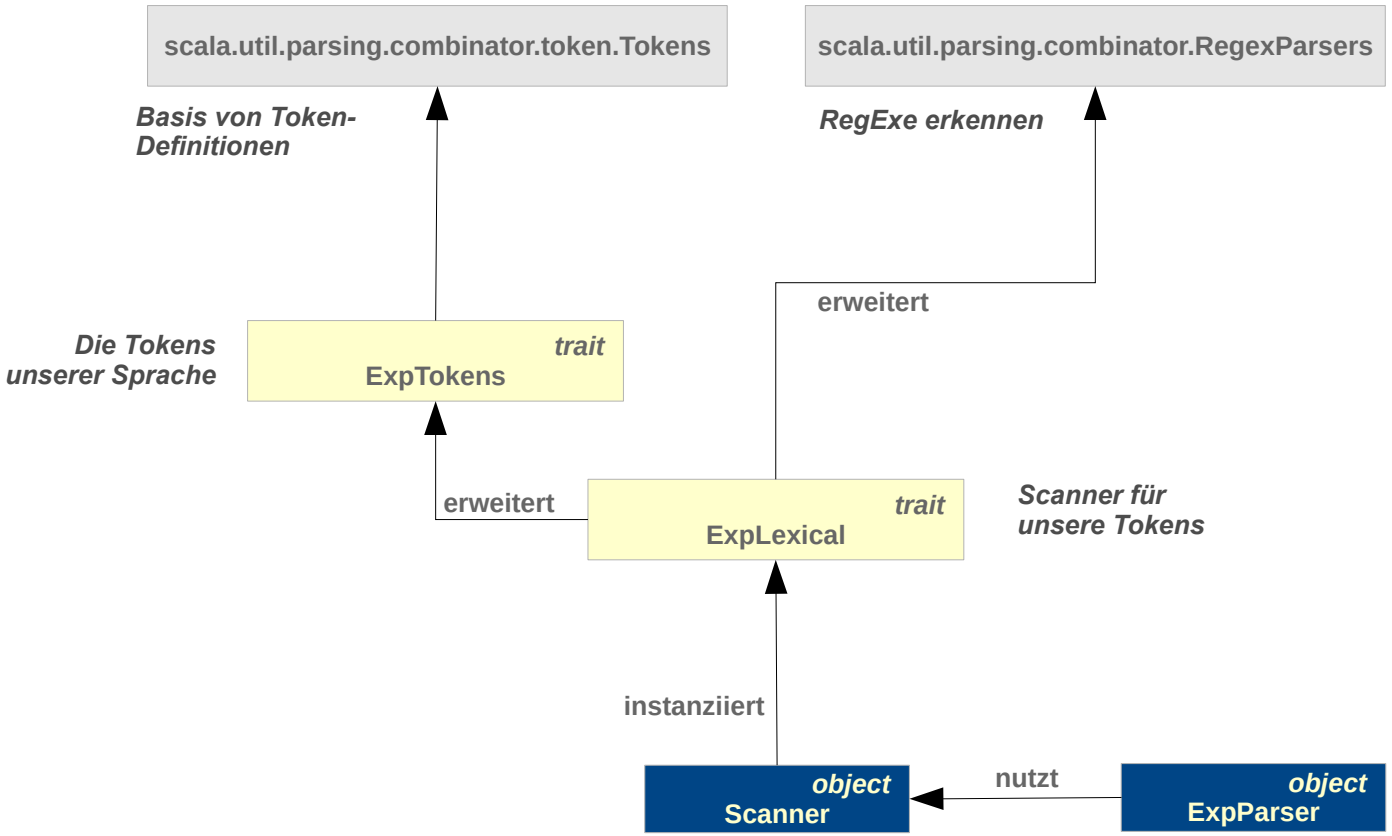
Parser-Kombinatoren sind Parser-Funktionen die „man sich selbst zusammenbauen kann“

Später werden diese genauer besprochen

Scanner-Implementierung mit Parser-Kombinatoren

Ein Scanner, der auf den Parser-Kombinatoren von Scala basiert

Übersicht



Scanner-Implementierung

Ein Scanner, der auf den Parser-Kombinatoren von Scala basiert (1)

```
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"
  }

}
```

Scanner-Implementierung

Ein Scanner, der auf den Parser-Kombinatoren von Scala basiert (2)

```
import scala.util.parsing.combinator.RegexParsers

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

}
```

Diese Konstruktion wird später erklärt.

Scanner-Implementierung

Ein Scanner, der auf den Parser-Kombinatoren von Scala basiert (3)

```
object ExpParser {
  def parse(code: String): ExpTree = {
    object Scanner extends ExpLexical {
      val tokenList: List[ExpToken] = tokenize(code)
      var pos = -1
      def nextToken: ExpToken = {
        pos = pos+1
        tokenList(pos)
      }
    }
    import Scanner._
    def parseOperation: Operation = {
      val exp1 = parseExp
      val opTok = Scanner.nextToken
      opTok match {
        case OperatorToken(opSymbol) => {
          val exp2 = parseExp
          val rightpTok = Scanner.nextToken
          rightpTok match {
            case RightPToken(_) =>
            case _ => throw new IllegalArgumentException("Expected " + RightPToken(")") + " found " + rightpTok)
          }
          opSymbol match {
            case "+" => Operation(exp1, Operator('+'), exp2)
            case "-" => Operation(exp1, Operator('-'), exp2)
            case "*" => Operation(exp1, Operator('*'), exp2)
            case "/" => Operation(exp1, Operator('/'), exp2)
            case _ => throw new IllegalArgumentException("Expected operator char, found " + opSymbol)
          }
        }
        case _ => throw new IllegalArgumentException("Expected operator, found " + opTok)
      }
    }
    parseExp
  }
}
```

Das ist unser guter alter Recursive-Descent-Parser !

```
def parseExp: ExpTree = Scanner.nextToken match {
  case NumberToken(x) => Number(x.toInt)
  case LeftPToken(_) => parseOperation
  case _ => throw new IllegalArgumentException
}
```

abstrakte Syntax

```
sealed abstract class ExpTree
case class  Number(v: Int) extends ExpTree
case class  Operation(exp1: ExpTree, op: Operator, exp2: ExpTree) extends ExpTree
```

Scanner-Implementierung – real

Der Scanner wird für einen praktischen Einsatz

- Eine Fehlerbehandlung beinhalten
- Ausführlichere Informationen über die Token liefern, Zumindest:
  - Typ
  - String
  - Position im Text (Ziele, Spalte)

Für „großen Sprachen“

- sind die Scanner meist handgeschrieben

  Achtung: Der Scanner ist sehr relevant für die Geschwindigkeit eines Compilers

Für „kleine Sprachen“

- wird traditionell der Einsatz von Scanner-Generatoren empfohlen
- mit der heutigen Unterstützung von regulären Ausdrücken ist ein handgeschriebener Scanner mit Nutzung der Regex-API eine sinnvolle Alternative
- Parser-Kombinatoren können die Parser- und Scanner-Generatoren ersetzen
- Scanner-loses Parsen (keine Trennung von Scanner und Parser) ist heute ebenfalls eine gängige Alternative