ISA
Institut für
SoftwareArchitektur

TECHNISCHE HOCHSCHULE MITTELHESSEN

Compilerbau CS1019

Th. Letschert

TH Mittelhessen Gießen

University of Applied Sciences

Reguläre Ausdrücke

- Reguläre Ausdrücke: Syntax und Semantik
- Transformation regulärer Ausdrücke in endliche Automaten
- Reguläre Ausdrücke als eingebettete DSL

Überblick

Some people, when confronted with a problem, think "I know, I'll use regular expressions." Now they have two problems.
(Programmierer-Folklore)

Reguläre Ausdrücke

Was ist das 1: *Kurze kryptische Texte zur Beschreibung von regulären Textstrukturen, also Muster, die ohne die Verwendung von Rekursion definiert werden können.*

Was ist das 2: *Ein Konzept, das wieder einmal beweist, dass nichts so praktisch ist wie eine gute Theorie.*

Grundlagen

- Syntax: wie sind sie aufgebaut
- Semantik: Was bedeuten sie

Reguläre Ausdrücke und endliche Automaten

Reguläre Ausdrücke sind äquivalent zu NEAs

Reguläre Ausdrücke in der Praxis

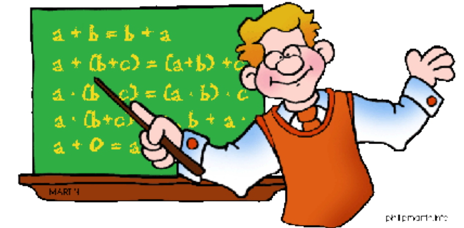
Reguläre Ausdrücke sind ein Standard-Werkzeug der Programmierer.

Ihre Notation ist umfangreicher als die Basisnotation aber in allen Sprachen in etwa gleich

```
sub deleteDoubleDots($) {
 while($_[0] =~ m/\.\./) {
    $_[0] =~ s/\/[^\/]*\/\.\.//;
 }
}
```

Reguläre Ausdrücke in Perl, der ersten Programmiersprache mit integrierten regulären Ausdrücken.

Reguläre Ausdrücke: Eine Muster-Algebra



Algebra / algebraische Struktur

Menge von Werten mit Verknüpfungsoperationen

Beispiele:

- Elementare Algebra: Zahlen den Rechenoperationen +, - *, /
- Algebraische Strukturen: Mathematik unterscheidet Algebren (algebraische Strukturen) mit bestimmte Eigenschaften: Gruppen, Ringe, Körper

Algebra: Wir arbeiten mit Ausdrücken (geschlossenen Formeln). Wir müssen keine Gleichungen lösen.

Textmuster als Algebra

Muster können „algebraisch“ definiert werden – statt mit Automaten

Dazu benötigt man

- Atomare Ausdrücke (vergleichbar zu 1, 2, 3, …)
- Verknüpfungsoperationen (vergleichbar zu +, -, *)

Reguläre Ausdrücke

Eine allgemein gebräuchliche „Muster-Algebra“,

- mit mehr oder weniger fest definierten atomaren Werten und Verknüpfungsoperationen,
- Die in vielen Programmiersprachen als Teil der API oder (gelegentlich) als Sprachbestandteil implementiert wurde,
- die jede/r Informatiker/in kennen muss !

Bildquelle: http://www.clipartpanda.com/categories/algebra-teacher-clipart

Reguläre Ausdrücke: Syntax

Die Menge der regulären Ausdrücke über einem Alphabet A (Menge von Zeichen) ist induktiv definiert:

- **Basis**
  - a für jedes $a \in A$ ist ein regulärer Ausdruck
  - ε ist ein regulärer Ausdruck
- **Operatoren**

  Sind r, r_1 und r_2 reguläre Ausdrücke dann auch
  - $r_1 \mid r_2$
  - $r_1 r_2$
  - r *

Beispiel: (ab | c)*

– **Notation: Assoziativität und Präzedenzen**

  Um eine vollständige Klammerung der regulären Ausdrücke vermeiden zu können, werden – wie bei arithmetischen Ausdrücken – Vorrangregeln (Präzedenzen) und Assoziativitäten definiert:
  - $r_1 \mid r_2 \mid r_3 = (r_1 \mid r_2) \mid r_3$ | hat die niedrigste Präzedenz und ist linksassoziativ
  - $r_1 r_2 r_3 = (r_1 r_2) r_3$ die Verkettung hat mittlere Präzedenz und ist linksassoziativ
  - $r_1 r_2{}^* = r_1 (r_2)^*$ * hat die höchste Präzedenz

Reguläre Ausdrücke: Semantik

Ein regulärer Ausdruck **bedeutet / steht für / definiert / repräsentiert** ein **Muster**

Ein Muster wird mit der **Menge der Wörter**, die zu ihm passen, identifiziert

Die **Semantik** (Bedeutung) eines regulären Ausdrucks ist darum eine Menge von Wörtern (Zeichenketten)

Die Semantik $\mathcal{L}(r)$ eines regulären Ausdrucks r über einem Alphabet A ist rekursiv die induktive Struktur von r definiert:

$\mathcal{L}(reg)$ =

- $\{a\}$ — falls $reg = a$ für irgendein $a \in A$
- $\{\}$ — falls $reg = \varepsilon$
- $\mathcal{L}(r_1) \cup \mathcal{L}(r_1)$ — falls $reg = r_1 \mid r_2$
- $\{ x\, y \mid x \in \mathcal{L}(r_1),\ y \in \mathcal{L}(r_2) \}$ — falls $reg = r_1\, r_2$
- $\{ \} \cup \mathcal{L}(r) \cup \mathcal{L}(r\, r) \cup \mathcal{L}(r\, r\, r) \ldots = \bigcup_{n \in N} \mathcal{L}(r^n)$ — falls $reg = r^*$

Beispiele:

Syntax r	Semantik $\mathcal{L}(r)$
ab	{ab}
ab | c	{ab, c}
ab*	{a, ab, abb, abbb, … }
(ab | c)*	{ab, c, abab, abc, cab, cc, ababab, … }

Reguläre Ausdrücke: Semantik

Weitere Beispiele:

Syntax	Semantik
0*10*	{w | w enthält eine 1 }
(1|(0(11*))*	{w | jede 0 in w wird von mindestens einer 1 gefolgt}
(0|1)*	{w | w ist eine beliebig lange Folge von 0en und 1en }
(0|1)*11(1|01)*	{w | w ist eine beliebig lange Folge on 0en und 1en, die mit 11 endet und die von einer weiteren Folge von 0en und 1en gefolgt wird, in der jede 0 von einer 1 folgt wird.}

Kann die Semantik (Bedeutung, gemeinte Menge) von einer Funktion berechnet werden?

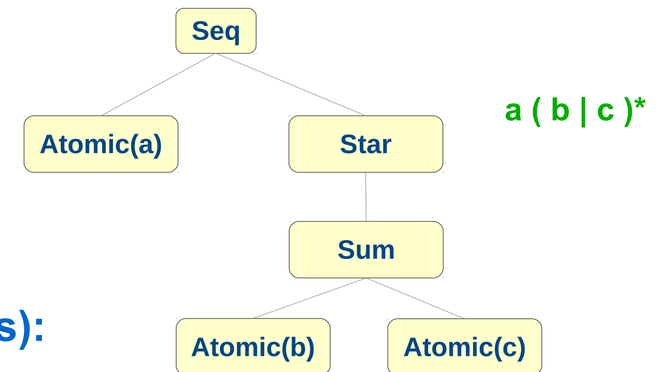
Reguläre Ausdrücke: Semantik

Semantik – Wir implementieren einfach die Definition.

Die Definitionen als Scala-Klassen / - Objekte

Die Klasse der regulären Ausdrücke über dem Alphabet der Zeichen (abstrakte Syntax):

```
sealed abstract class RegEx
case class Atomic(a: Char) extends RegEx
case object Empty extends RegEx
case class Sum(r1: RegEx, r2: RegEx) extends RegEx
case class Seq(r1: RegEx, r2: RegEx) extends RegEx
case class Star(r: RegEx) extends RegEx
```



… und ihre Interpretation $\mathcal{L}$(reg) als Menge von Worten (Strings):

```
def L(reg: RegEx) : Set[String] = reg match {
    case Atomic(x) => Set(x.toString)
    case Empty => Set("_")
    case Seq(r1, r2) =>
      for (s1 <- L(r1);
           s2 <- L(r2) ) yield s1 ++ s2
    case Sum(r1, r2) => L(r1) ++ L(r2)
    case Star(r) => L(Empty) ++
                    L(r) ++
                    L(Seq(r, r)) ++
                    L(Seq(r, r)) ++
                    L(Seq(r, Seq(r, r))) ++
                    . . . etc . . .
}
```

"_" damit das leere Wort sichtbar wird.

Der Stern-Operator bezeichnet eine unendlich große Menge.

$\mathcal{L}(r^n) = \{\ \} \cup \mathcal{L}(r) \cup \mathcal{L}(r\ r) \cup \mathcal{L}(r\ r\ r) \ldots = \cup_{n \in N} \mathcal{L}(r^n)$

Hier müssten darum unendlich viele weitere Fälle aufgeführt werden. - Was natürlich nicht geht.

Reguläre Ausdrücke: Semantik

Die unendlichen Mengen können als Streams* repräsentiert werden:

```
def L(reg: RegEx) : Stream[String] = reg match {
    case Atomic(x) => Stream(x.toString)
    case Empty => Stream("_")
    case Seq(r1, r2) =>
      for (s1 <- L(r1);
           s2 <- L(r2) ) yield s1 ++ s2
    case Sum(r1, r2) => L(r1) #::: L(r2)
    case Star(r) => {
      def star(v: RegEx): Stream[String] =  L(v) #::: star(Seq(r, v))
      star(Empty)
    }
}
```

#::: Verknüpfung von zwei Strömen

Ein Strom ist eine unendliche Sequenz, deren Elemente aber nur auf Aufforderung hin generiert werden

Der Stern-Operator bezeichnet eine unendlich große Menge:

$\mathcal{L}(r^n) = \{\,\} \cup \mathcal{L}(r) \cup \mathcal{L}(r\ r) \cup \mathcal{L}(r\ r\ r) \ldots = \cup_{n \in N} \mathcal{L}(r^n)$

$\mathcal{L}(r^*)$ wird hier rekursiv definiert:

$\mathcal{L}(r^*) = \text{Star}(\varepsilon)$

Mit $\text{Star}(v) = \mathcal{L}(v) \cup \text{Star}(r\ v)$

Star ist endlos-rekursiv. Die Endlos-Rekursion wird durch den Strom handhabbar.

*** https://www.scala-lang.org/api/current/scala/collection/immutable/Stream.html**

Reguläre Ausdrücke: Semantik

Beispiel:

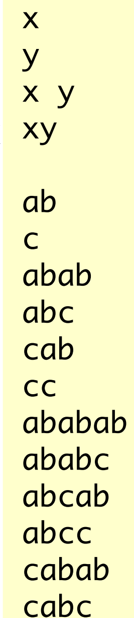
```
L(Atomic('x')).foreach{ println(_) }

L(Atomic('y')).foreach{ println(_) }

L(Sum(Atomic('x'), Atomic('y'))).foreach{ v => print(v + " ") }
println()

L(Seq(Atomic('x'), Atomic('y'))).foreach{ v => print(v + " ") }
println()

L(Star(Sum(Seq(Atomic('a'),Atomic('b')),Atomic('c')))).foreach{ println(_) }
```



L generiert alle Wörter (Strings), die zu einem regulären Ausdruck passen.

Reguläre Ausdrücke: Semantik

Ausdruckskraft regulärer Ausdrücke:

Reguläre Ausdrücke haben die gleiche (beschränkte) Ausdruckskraft wie endliche Automaten:

Mit ihnen können nur nicht-rekursive Muster dargestellt werden.

Beispiel:

Das Muster

Eine beliebig lange Folge von '(' gefolgt von einer gleich langen Folge von ')'

erfordert einen Mechanismus zum unbeschränkten Zählen

- **Ein Automat modelliert alles in Zuständen, auch den Zählerstand. Ein endlicher Automat hat eine feste endliche Zahl möglicher Zustände, kann darum also nicht beliebig (unbeschränkt) weit zählen.**
- **Ein regulärer Ausdruck ist ein „algebraischer Ausdruck", also eine „geschlossene Formel" ohne Rekursion. Die direkte oder indirekte rekursive Definition eines Musters wie beispielsweise**

  M = '(' ')' | '(' M ')'

  ist darum kein regulärer Ausdruck.

'(' ')' | '(' M ')'	*kein reg. Ausdruck, M undefiniert*
M = '(' ')' | '(' M ')'	*kein reg. Ausdruck, M rekursiv definiert*
M = ')' | M ')'	*kein reg. Ausdruck, M rekursiv definiert, aber*
M = ')' ')' *	*äquivalente Definition (gleiche Sprache) mit reg. Ausdruck*

Reguläre Ausdrücke: Mustererkennung – naiver Algorithmus

Reguläre Ausdrücke

sind eine Notation zur Definition von Mustern

Die i.d.R. unendlichen Mengen

von Zeichenketten, die einem regulären Ausdruck zugeordnet sind, repräsentieren das „gemeinte“ Muster, sind aber als <u>Mengen</u> eher uninteressant. Der Test, ob ein Wort dazugehört, ist das Interessante.

Ein naiver Algorithmus zur Mustererkennung

Auf Basis eines regulären Ausdrucks muss man testen, ob ein vorgelegtes Wort w zu der Menge gehört, die das Muster definiert wird:

```
def patternMatch(pattern: RegEx, w: String): Boolean = {
  def L(reg: RegEx) : Stream[String] = reg match {
      case Atomic(x) => Stream(x.toString)
      case Empty => Stream("")
      case Seq(r1, r2) =>
        for (s1 <- L(r1);
             s2 <- L(r2) ) yield s1 ++ s2
      case Sum(r1, r2) => L(r1) #::: L(r2)
      case Star(r) => {
        def  star(v: RegEx): Stream[String] =  L(v) #::: star(Seq(r, v))
        star(Empty)
      }
  }
  L(pattern).contains(w)
}
```

Prüfe ob das Wort zu der definierten Menge gehört.
Dieses Verfahren funktioniert leider nicht! Warum nicht?

Reguläre Ausdrücke: Mustererkennung – naiver Algorithmus

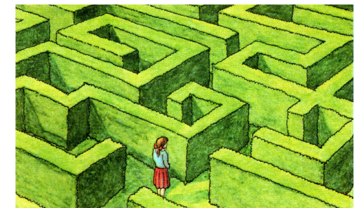
Naiver Algorithmus zur Mustererkennung

Generiere die Worte (Texte), die durch das Muster definiert wurden;
prüfe dann, ob das vorgelegte Wort in der Menge enthalten ist.

Der naive Algorithmus zur Mustererkennung ist unbrauchbar

- Da das durch einen regulären Ausdruck definierte Muster i.d.R. eine unendlich große Menge bezeichnet,
- kann ein Test, der die Menge nach einem vorgelegten Wort durchsucht, nur dann terminieren, wenn das Wort in der Menge enthalten ist.
- Nicht einmal dann, wenn das Muster zum Wort passt, ist garantiert, dass der Algorithmus terminiert. Die Suche kann sich ja in unendlich vielen „falschen" Pfaden verlieren, ohne je zum gesuchten und vorhandenen Wort zu kommen.

In einem unendlich großen Irrgarten kann man sich selbst dann für immer verlaufen, wenn es einen Weg zum Ausgang gibt.



Reguläre Ausdrücke und endliche Automaten

Endliche Automaten sind „irgendwie das Gleiche wie“ reguläre Ausdrücke

- Reguläre Ausdrücke und Automaten sind ein Formalismus zur Definition von Mustern.
- Allerdings sind endliche Automaten weniger „handlich“ als reguläre Automaten
  - Automaten kann man nicht hinschreiben, man muss sie zeichnen
  - Deterministische Automaten sind nicht so leicht zu definieren.
- Für deterministische endliche Automaten gibt es einfache und effiziente Implementierungen und zudem gibt es einen einen Algorithmus um nicht-deterministische in deterministische Automaten zu transformieren.
- Glücklicherweise **kann man jeden regulären Ausdruck in einen nichtdeterministischen endlichen Automaten transformieren** (und umgekehrt).
- Implementierung einer effizienten Mustererkennung:

  Regulärer Ausdruck
  => NEA
  => DEA
  => DEA-Implementierung

Automaten mit ε-Transitionen

Für die Transformation

von regulären Ausdrücken in Automaten werden Automaten mit sogenannten ε-Transitionen benötigt:

- Reguläre Ausdrücke können in Automaten mit ε-Transitionen umgewandelt werden und
- diese dann in Automation ohne ε-Transitionen.

Eine ε-Transitionen

ist ein „spontaner“ Zustandsübergang, d.h. ein Zustandsübergang bei dem

- kein Zeichen oder das „leere Zeichen“

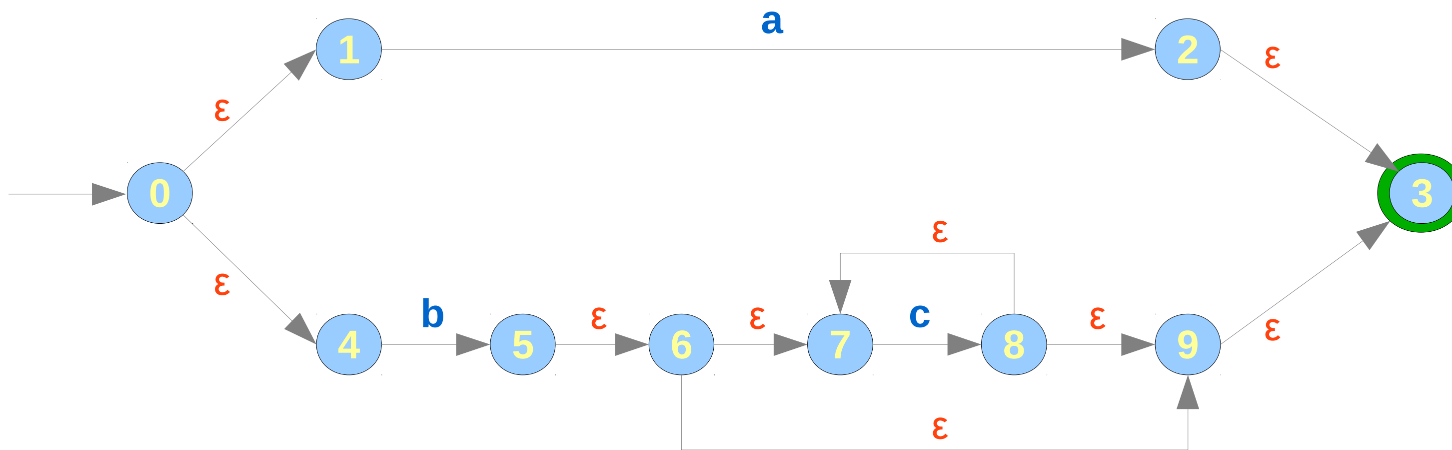
erkannt / konsumiert wird.

Der Pfad wir mit einem ε markiert, dieses ist aber im Wort „unsichtbar“.

Automaten mit ε-Transitionen

Beispiel

Automat mit ε-Transition, äquivalent zum regulären Ausdruck a | bc*



Alle Pfade von 1 nach 3 sind mit
a , b, bc, bcc , . . .
Markiert. Die Äquivalenz zum regulären Ausdruck ist offensichtlich.

Vom regulären Ausdruck zum Automaten mit ε-Transitionen

Transformations-Algorithmus

Der Algorithmus zur Transformation eines regulären Ausdrucks r in einen Automaten wird rekursiv über die Struktur des Ausdrucks definiert. (Thompson's Konstruktion*)

Basis: Der reg. Ausdruck r ist der leere Ausdruck, ε oder ein Symbol a

– $\mathcal{A}(\emptyset)$ =

Enthält keinen akzeptierenden Pfad

– $\mathcal{A}(\varepsilon)$ =

Geht spontan in den akzeptierenden Zustand über

– $\mathcal{A}(a)$ =

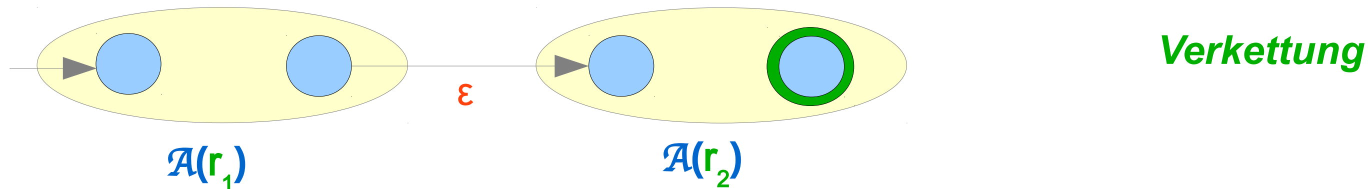
Geht mit a in den akzeptierenden Zustand über

*nach Ken Thompson: *Regular Expression Search Algorithm* CACM 11:6 (1968)

Vom regulären Ausdruck zum Automaten mit ε-Transitionen

Induktionsschritt : Reg. Ausdruck wurde mit Alternative (|), *Verkettung* oder Stern (*) gebildet

– $\mathcal{A}(r_1 r_2)$ =



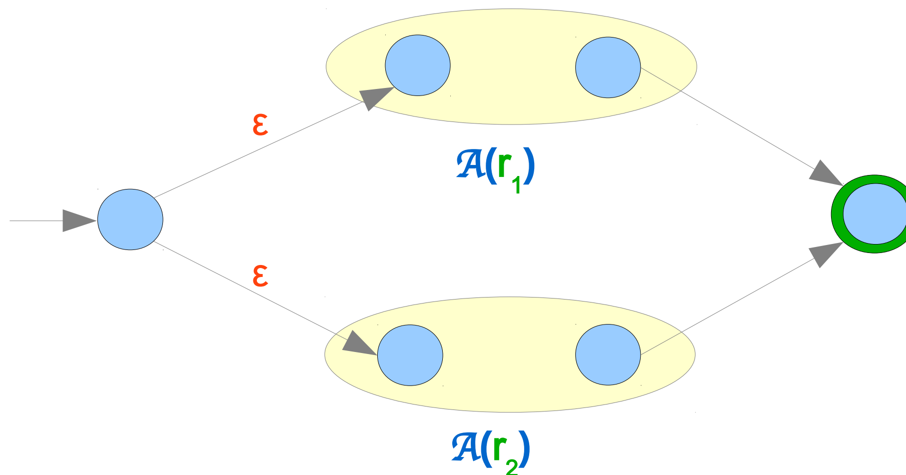
Verkettung

Startzustand ist der Startzustand des ersten Automaten, akzeptierender Zustand ist der akzeptierende Zustand des zweiten Automaten. Zwischen dem akzeptierenden Zustand des ersten und dem Startzustand des zweiten Automaten wird eine ε-Transition eingefügt.

Vom regulären Ausdruck zum Automaten mit ε-Transitionen

Induktionsschritt : Reg. Ausdruck wurde mit |, Verkettung oder * gebildet

– $\mathcal{A}(r_1 | r_2)$ =



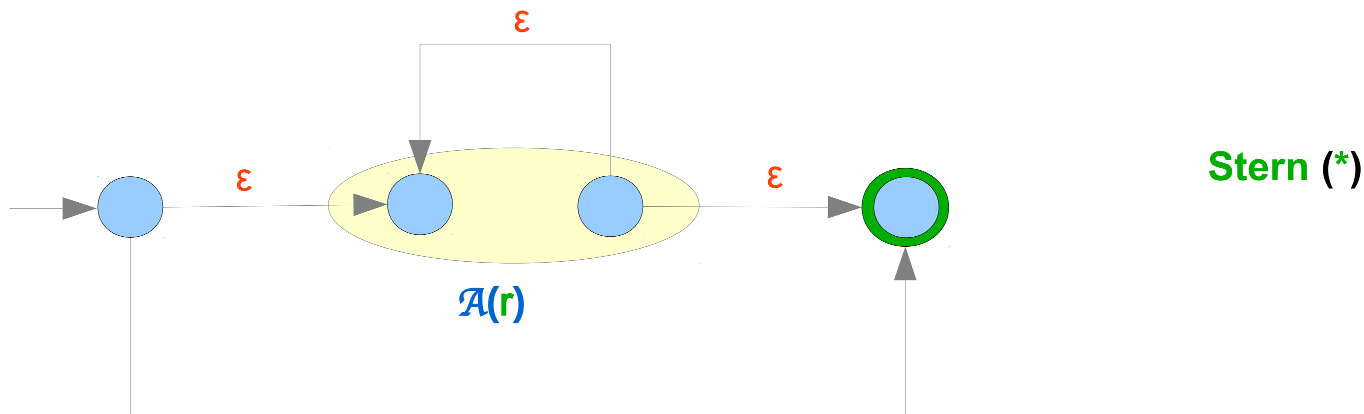
Alternative (|)

Ein Startzustand und ein akzeptierender Zustand werden hinzugefügt. Von neuen Startzustand führen zwei ε-Transitionen zu den Startzuständen der gegeben Automaten. Von deren akzeptierenden Zustand wird jeweils eine ε-Transition eingefügt, die zum neuen akzeptierenden Zustand führt.

Vom regulären Ausdruck zum Automaten mit ε-Transitionen

Induktionsschritt : Reg. Ausdruck wurde mit |, Verkettung oder * gebildet

– $\mathcal{A}(r^*)$ =



Stern (*)

Ein Startzustand und ein akzeptierender Zustand werden hinzugefügt. Von neuen Startzustand führen eine ε-Transitionen zum Startzustand des gegeben Automaten. Von dessen akzeptierenden Zustand führt eine ε-Transition zum neuen akzeptierenden Zustand. Von akzeptierenden Zustand des gegeben Automaten führt eine ε-Transition zurück zu dessen Startzustand. Mit einer zusätzlichen ε-Transition kann der gegebene Automat komplett umgangen werden.

Elimination von ε-Transitionen

Ein Automat mit ε-Transitionen

kann in einen äquivalenten Automaten ohne ε-Transitionen transformiert werden.

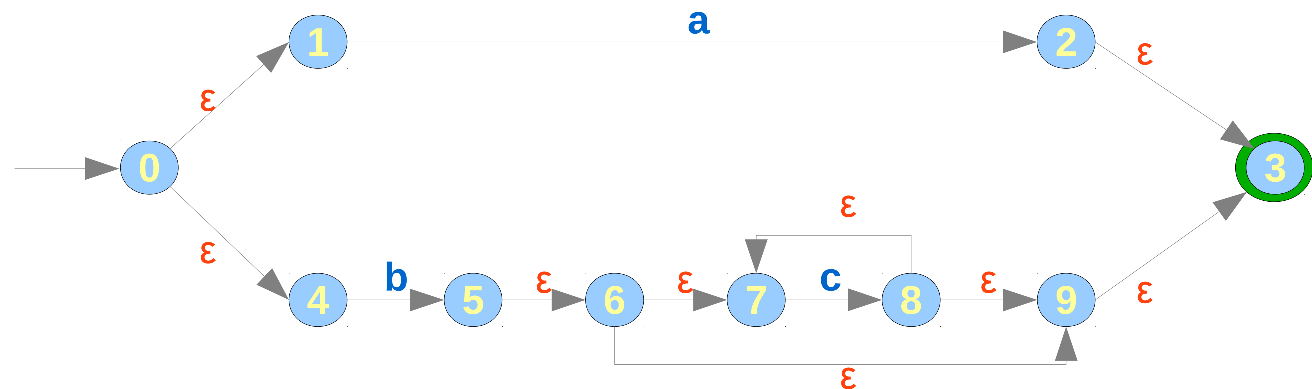
Transformations-Idee

Ist man in einem Zustand mit ε-Transitionen, dann ist man „eigentlich“ auch schon in allen Zuständen, die sich über ε-Transitionen erreichen lassen.

ε-Erreichbarkeit

Mit einer Tiefensuche kann – ausgehend von jedem Zustand – die Menge der Zustände berechnet werden, die über ε-Transitionen erreichbar sind.

Beispiel:



0 ~> 1, 4
5 ~> 6, 7, 9, 3
6 ~> 9, 3

8 ~> 7, 9, 3
9 ~> 3
2 ~> 3

Elimination von ε-Transitionen

Mit der ε-Erreichbarkeit

kann ein äquivalenten Automat ohne ε-Transitionen konstruiert werden.

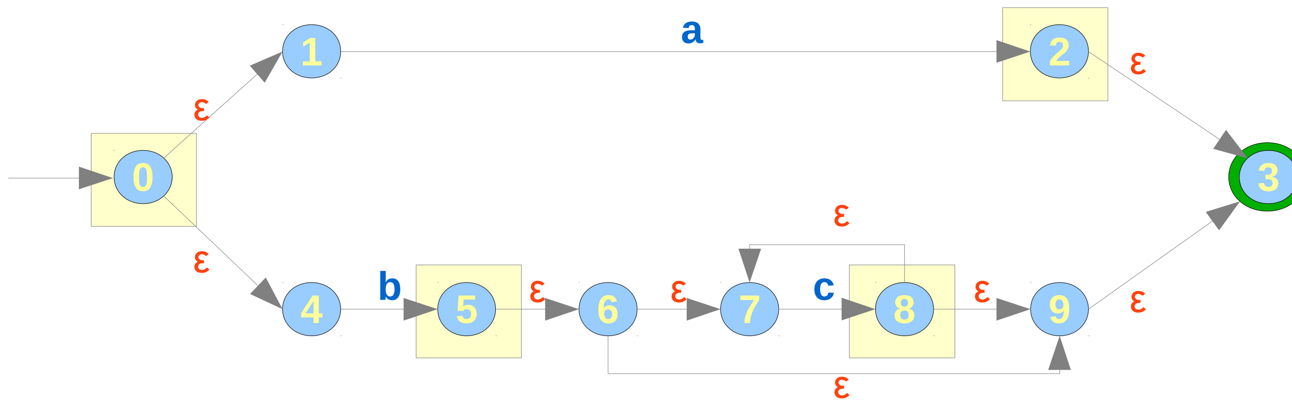
Konstruktions-Idee

- Alle Folgen von ε-Transitionen, gefolgt von einer „richtigen“ Transition mit einem Zeichen a werden zu einer a-Transition zusammen gefasst.
- Nur diese Transitionen entsprechen echten Symbolen und und nur solche Transitionen enden in interessanten Zuständen
- Der neue Automat enthält nur interessante Zustände und zusammengefasste Transitionen.
  Natürlich ist der Startzustand ebenfalls interessant.
- Akzeptierende Zustände im neuen Automaten sind alle interessanten akzeptierenden Zustände im alten Automaten plus alle Zustände, die mit ε-Transitionen von diesen aus erreichbar sind.

Elimination von ε-Transitionen

Beispiel: Automat für a | bc* mit ε-Transitionen,

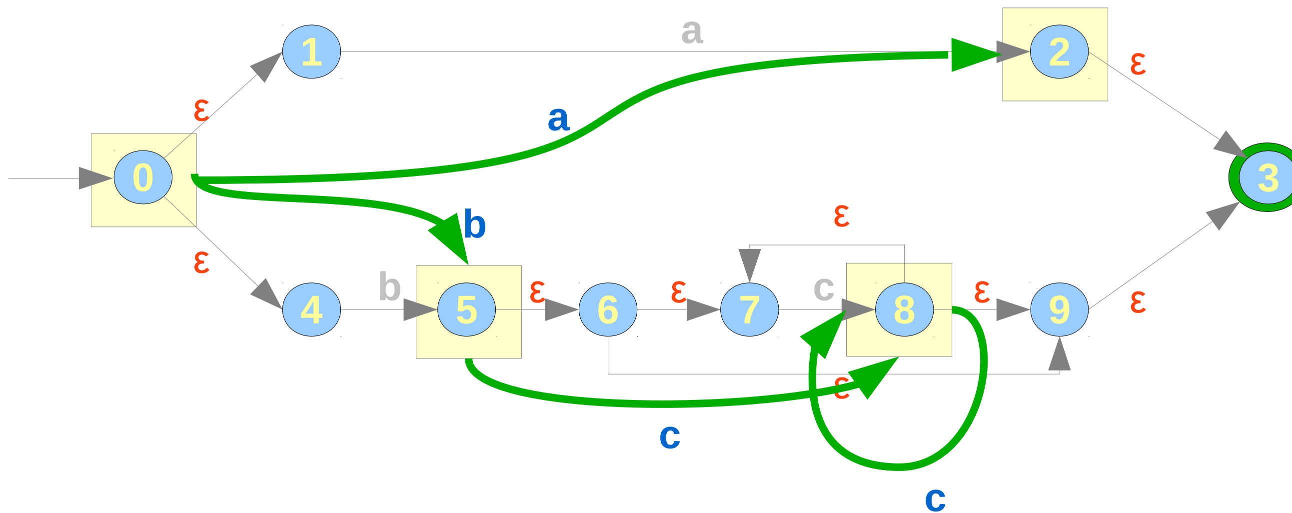
Interessante Zustände



Elimination von ε-Transitionen

Beispiel Automat für **a | bc*** mit ε-Transitionen,

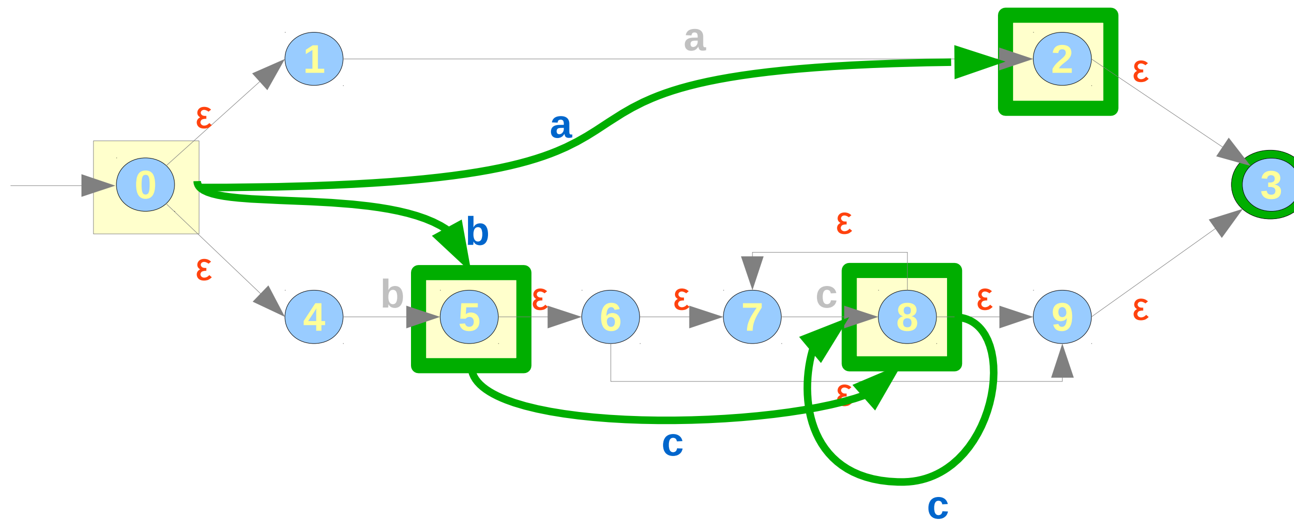
Interessante Zustände und zusammengefasste Transitionen (→)



Elimination von ε-Transitionen

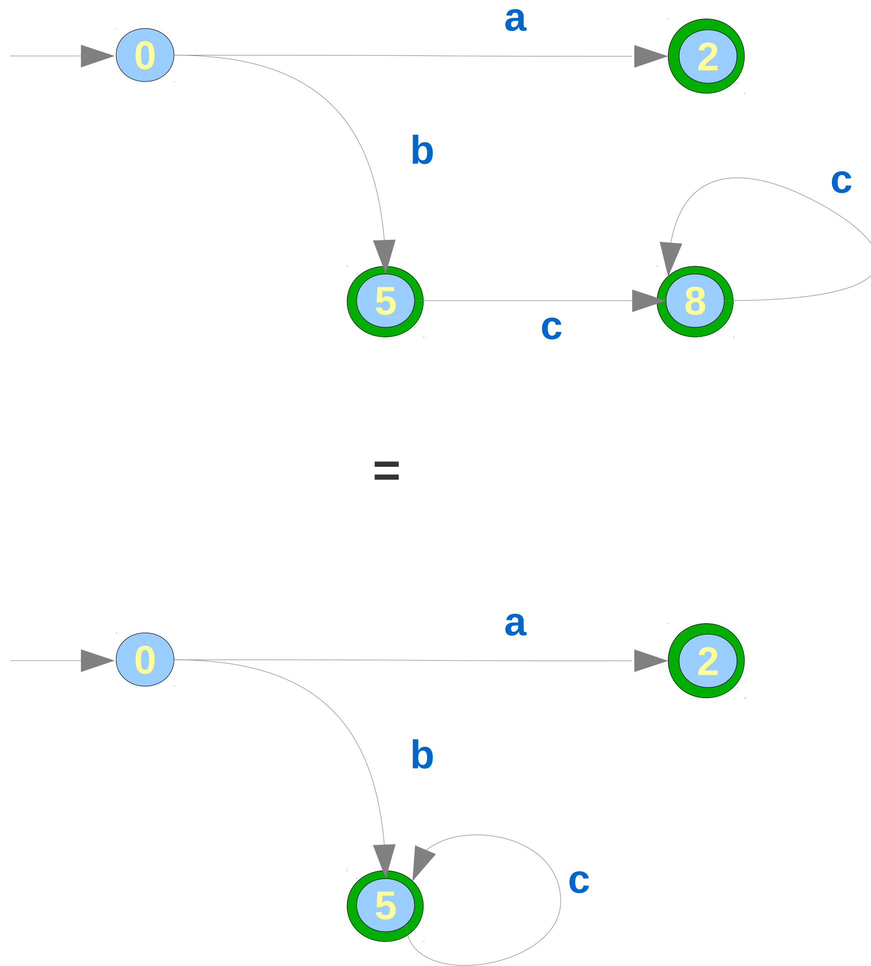
Beispiel Automat für **a | bc*** mit ε-Transitionen,

Interessante Zustände, zusammengefasste Transitionen und akzeptierende Zustände



Elimination von ε-Transitionen

Beispiel Automat für a | bc* ohne ε-Transitionen,



Bereinigte Version von vorheriger Folie

Reguläre Ausdrücke: Mini-DSL in vielen Programmiersprachen

Pioniere: Unix und Perl

- Beginnend mit Unix-Scriptsprachen und der Programmiersprache Perl sind reguläre Ausdrücke heute in viele Programmiersprachen integriert.

DSL – *Domain Specific Language*

- DSL: Eine „kleine“ Programmiersprache, für ein spezielles Anwendungsgebiet.
- Beispiel: HTML, SQL, Formel-Notation bei Spredsheets (excel), …
  - Interne DSL: Spezialsprache ist in eine andere Sprache eingebettet
  - Externe DSL: Spezialsprache wird mit einen Tools ohne Unterstützung durch eine „Wirtssprache“ genutzt

Reguläre Ausdrücke

- Weitverbreitete interne DSL zur Mustererkennung bei Texten
- Glücklicherweise in allen „Wirtssprachen“ mit im Wesentlichen gleicher Syntax.

Reguläre Ausdrücke in Programmiersprachen

Reguläre Ausdrücke in Unix-Notation

Man unterscheidet Sonderzeichen und normale Zeichen. Sonderzeichen sind z.B. : `* + [ ] ? ^ ( ) . $`

- ein „normales“ Zeichen steht für sich selbst
- . steht für ein beliebiges Zeichen außer `'\n'`
- Falls ein Sonderzeichen nicht als solches interpretiert werden soll, ist ein „\“ voranzustellen
- Auch in Apostrophe eingeschlossene Strings werden verbatim interpretiert.
- Klammerung erfolgt durch (und)
- Konkatenation zweier reg. Ausdrücke erfolgt ohne expliziten Operator
- Alternativen werden mittels | gebildet
- ein nachgestellter * steht für beliebige Wiederholung
- ein nachgestelltes **+** steht für nichtleere Wiederholung
- ein nachgestelltes **?** bezeichnet einen optionalen Anteil
- ^ am Anfang eines regulären Ausdrucks steht für Zeilenanfang
- **$** am Ende eines regulären Ausdrucks steht für Zeilenende
- eine **Zeichenklasse** steht für ein Zeichen.

  Sie kann durch **Zeichen-Aufzählung** $x_1x_2\ldots x_n$ und **Bereichsangaben** $x_1 - x_n$ gebildet werden:

  - $x_1 - x_n$ steht für ein Zeichen aus dem Bereich, z.B. `[0-9]`
  - $x_1x_2\ldots x_n$ steht für ein Zeichen aus der Menge der angegebenen Zeichen, z.B. `[abcx]`
  - beide Schreibweisen können kombiniert werden z.B. `[0-9a-zA-Z_]`
  - Eine ^-Zeichen am Anfang einer Zeichenklasse `[^...]` spezifiziert die komplementäre Zeichenmenge, z.B. steht `[^0-9]` für ein beliebiges Zeichen außer einer Ziffer

Aus: Michael Jäger:
Compilerbau – eine Einführung SS 2015; 13. April 2015 (Version 3.3.0 vom 13.4.2015)

Reguläre Ausdrücke in Java und Scala

Reguläre Ausdrücke in Java

Syntax im wesentlichen gleich zur Unix-Notation – siehe API-Doku für Details

Einige Beispiele.

- `x` jedes Zeichen ist ein Muster das für sich selbst steht
- `[xy]` eine Klasse, ein `x` oder ein `y`
- *regEx1* | *regex2* ein Text der zu *regEx1* oder *regEx2* passt
- `\\d` eine Ziffer (= `"[0123456789]"` = `"[0-9]"`)
- `\\s` ein weißes Zeichen
- `\\w` ein Wortzeichen (Buchstabe, Ziffer oder Unterstrich)
- `.` ein einziges beliebiges Zeichen
- *regEx1** ein Text, der zu *regEx1* passt, beliebig oft wiederholt
- `\\.` ein Punkt – Sonderzeichen wie der Punkt brauchen „Escape“-Zeichen
- `^.` irgendein Zeichen am Anfang des Textes

Reguläre Ausdrücke in Java und Scala

Reguläre Ausdrücke in Java – einfaches Anwendungsbeispiel

```
public class MatchExample {

  public static void main(String[] args) {
    String[] strings = {
               "Abc die Katze lief im Schnee.",
               "Peter schlief im Klee.",
               "Klausi trinkt gerne Tee mit Ruhm.",
               "Die Katze liebt keinen See."};

    String[] pattern = {"Katze", ".*ee.*"};

    for (String s: strings) {
      System.out.println(s);
      for (String word : s.split("\\s")) {
         for (String pat: pattern) {
            if (word.matches(pat)) {
               System.out.println(
                  "\tenthält das Wort \""
                  + word + "\" das zu \""
                  + pat + "\" passt");
            }
         }
      }
    }
  }

}
```

```
Abc die Katze lief im Schnee.
     enthält das Wort "Katze" das zu "Katze" passt
     enthält das Wort "Schnee." das zu ".*ee.*" passt
Peter schlief im Klee.
     enthält das Wort "Klee." das zu ".*ee.*" passt
Klausi trinkt gerne Tee mit Ruhm.
     enthält das Wort "Tee" das zu ".*ee.*" passt
Die Katze liebt keinen See.
     enthält das Wort "Katze" das zu "Katze" passt
     enthält das Wort "See." das zu ".*ee.*" passt
```

Reguläre Ausdrücke in Java und Scala

Die Klassen Pattern und Matcher

- *Pattern*: „übersetzter“ regulärer Ausdruck
- *Matcher*: wird aus einem Pattern erzeugt und dient der Mustererkennung

Gruppen

- Im regulären Ausdruck durch Klammer abgegrenzte Teilmuster
- Können mit matcher.group identifiziert werden
- (?: …) klammert
- (…) klammert und definiert eine namenlose Gruppe
- (<name> …) klammert und definiert eine Gruppe mit Namen

Beispiel

Sätze der Form

... X liebt Y …
... X liebt den Y …
... X liebt die Y …
... X steht auf den Y ...
... X steht auf die Y …

erkennen

Reguläre Ausdrücke in Programmiersprachen

Beispiel zu

Pattern, Matcher

und Gruppen

```
String text = "Adele liebt Rosen. "
      + "Pauline liebt den Gerd. "
      + "Holger steht auf Mathe. "
      + "Karla liebt Hugo. "
      + "Brunhilde liebt die Mechthilde. "
      + "Hugo steht auf die Mathematik. "
      + "Die Rose liebt den Sonnenschein. ";

String regex = "((.* )|^)(?<wer>\\w+) (liebt|(steht auf)) ((den|die) )?(?<wen>\\w+).*";

Pattern pattern = Pattern.compile(regex);

Map<String, String> relation = new TreeMap<String, String>();

for (String sentence : text.split("\\.")) {
      Matcher matcher = pattern.matcher(sentence.trim());
      // Passt der "Satz" zum Muster ?
      if (matcher.matches()) {
      // die mit wer und wen markierten Gruppen einander zuordnen
      relation.put(matcher.group("wer"), matcher.group("wen")); }
}
for (String wer: relation.keySet()) {
      System.out.println(wer + " liebt " + relation.get(wer));
}
```

```
Adele liebt Rosen
Brunhilde liebt Mechthilde
Holger liebt Mathe
Hugo liebt Mathematik
Karla liebt Hugo
Pauline liebt Gerd
Rose liebt Sonnenschein
```

Siehe API-Doku zu *java.util.regex* !

Reguläre Ausdrücke in Java und Scala

Reguläre Ausdrücke in Scala

- Paket *scala.util.matching* (siehe API-Doku!)
- Basiert auf den Klassen der Java-API

Beispiel 1:

```
object Example1_Main extends App {
  val strings = List(
      "Abc die Katze lief im Schnee.",
      "Peter schlief im Klee.",
      "Klausi trinkt gerne Tee mit Ruhm.",
      "Die Katze liebt keinen See.")

  val pattern = List("Katze", ".*ee.*")

  for (s <- strings) {
    for (word <- s.split("\\s")) {
      println(s)
      for (pat <- pattern) {
        if (word.matches(pat)) {
          println( "\tenthält das Wort \""
                    + word + "\" das zu \""
                    + pat + "\" passt");
        }
      }

      }
  }
}
```

Entspricht 1:1 der Java-Variante oben

Reguläre Ausdrücke in Java und Scala

Reguläre Ausdrücke in Scala

– Verarbeitung von Match-Gruppen ist in die *Pattermatch*-Syntax integriert

Beispiel 2:

```
object Example2_Main extends App {

  val text = "Adele liebt Rosen. " +
      "Pauline liebt den Gerd. " +
      "Holger steht auf Mathe. " +
      "Karla liebt Hugo. " +
      "Brunhilde liebt die Mechthilde. " +
      "Hugo steht auf die Mathematik. " +
      "Die Rose liebt den Sonnenschein. "

  val pattern = """(?:(?:.* )|^)(\w+) (?:liebt|(?:steht auf)) (?:(?:den|die) )?(\w+).*""".r

  var relation: Map[String, String] = Map()

  for (sentence <- text.split("\\.")) {
    sentence match {
      case pattern(wer, wen) => relation += (wer -> wen)
      case _ =>
    }
  }
  for ((wer, wen) <- relation) {
      println(wer + " liebt " + wen);
  }
}
```

Entspricht der Java-Variante oben – ist nur wesentlich kompakter.

Rückblick

Ziel dieser Lerneinheit

- Sie wissen was reguläre Ausdrücke – inklusive *Matchgruppen* – sind und können sie ab sofort produktiv einsetzen (eventuell unter Zuhilfenahme von weiteren Online-Dokumenten).
- Sie kennen die Bezüge von endlichen Automaten und regulären Ausdrücken und haben eine Vorstellung
  - von ihrer Ausdruckskraft und
  - ihrer Implementierung
- Sie haben ein vertieftes Verständnis von Syntax und Semantik:
  - Was ist die Syntax und Semantik regulärer Ausdrücke
  - Syntaktische Konstrukte können Unendliches bedeuten
  - Die Bedeutung syntaktischer Konstrukte kann konkret berechnet werden:

    in Scala recht bequem und

    selbst dann, wenn dabei unendlich Großes entsteht
- Sie haben Ihr Verständnis der syntaktische Analyse gefestigt:

  Die Analyse eines Textes / Wortes (Mustervergleich)
  - ist die Suche nach dem Text / Wort in einer (meist) unendlichen Menge
  - die trotz der Unendlichkeit u.U. effizient ausgeführt werden kann