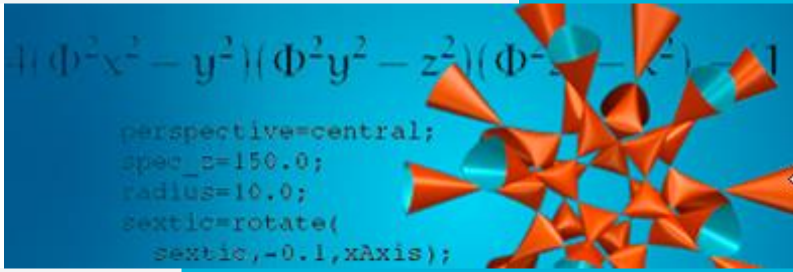
Theoretische Informatik: Reguläre Sprachen und reguläre Ausdrücke

Prof. Dr. Elmar Tischhauser



Reguläre Ausdrücke und reguläre Sprachen

1. Reguläre Sprachen
2. Syntaxdiagramme
3. Erweiterungen
4. Anwendungen

Reguläre Sprachen

Sei Σ ein Alphabet. Reguläre Sprachen über Σ
sind induktiv definiert

1. Regulär sind:
   - $\varnothing$ // die leere Sprache
   - $\{\varepsilon\}$ // enthält nur das leere Wort
   - $\{a\}$ für jedes $a \in \Sigma$.
2. Sind L und M reguläre Sprachen, dann auch
   - $L + M$ // $L \cup M$
   - $L \circ M$
   - $L^* = \bigcup_{n \geq 0} L^n$

- L regulär $\Leftrightarrow$ durch einen regulären Ausdruck beschreibbar

Reguläre Sprachen

Sei $\Sigma = \{a_1, a_2, \ldots, a_n\}$ ein Alphabet. Dann folgt, dass auch die folgenden Sprachen regulär sind:

- Σ - denn $\Sigma = \{a_1\} \cup \{a_2\} \cup \ldots \cup \{a_n\}$
- Σ^* - mit
  - $\{ w \}$ für jedes $w \in \Sigma^*$
    - denn für $w = w_1 w_2 \ldots w_n$ gilt $\{w\} = \{w_1\}\{w_2\}\ldots\{w_n\}$
- Jede endliche Teilmenge $L \subseteq \Sigma^*$

• Ist L regulär, dann auch

- $L^n = LL \ldots L$ (n-mal) für jedes $n > 0$
- $L^+ = LL^*$
- $L? = L \cup \{\varepsilon\}$
- $L^{\{m,n\}} = L^m(\{\varepsilon\} \cup L \cup L^2 \cup \ldots \cup L^{n-m})$
- $L^{\{m,*\}} = L^m L^*$

Beispiele

- Reguläre Sprachen sind z.B.:
  - $\{a, b, abba, \varepsilon\}$
  - $\{\ ba^n b \mid n \in \mathbb{N}\}$
  - $\{a^n ba \mid n \text{ gerade}\}$
  - $\{a^n b^m \mid n, m \in \mathbb{N}\}$
  - $\{a^n b^m \mid n + m \text{ gerade}\}$ … wieso ? …
  - Die Menge aller int Konstanten in Java { 0, -23,42, 1234,…}
- Keine regulären Sprachen sind
  - $\{\ a^n ba^n \mid n \in \mathbb{N}\ \}$
  - $\{a^n b^m \mid n \leq m\}$
- … wieso nicht ?
  - ... kommt später

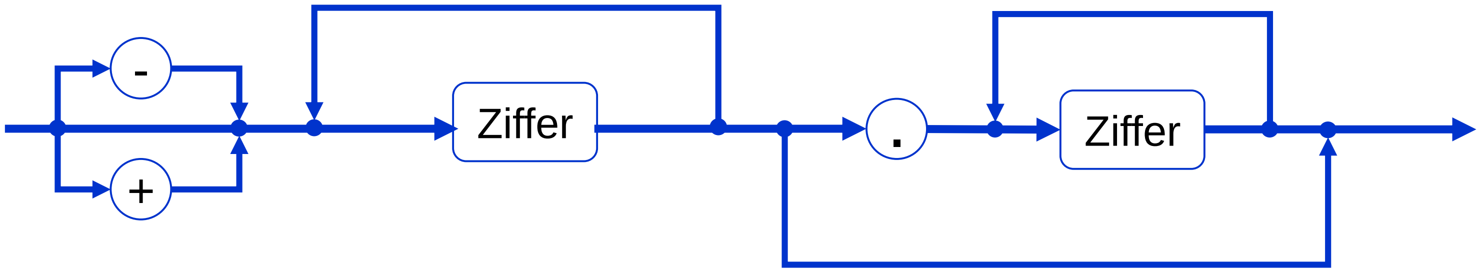
Spezifikation

- Was sind erlaubte Real-Zahlen in Pascal ?
  - erlaubt:
    - $3.14\,,\; 0.01, -1.0\,,\; 42$
  - nicht erlaubt:
    - $.21\,,\; 31., -0.$
  - Wie kann man eindeutig festlegen (spezifizieren), was erlaubt ist und was nicht ?

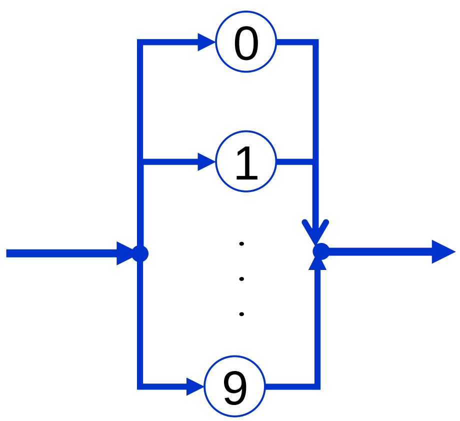
Syntaxdiagramme

- Bsp: Real Zahlen

Real:



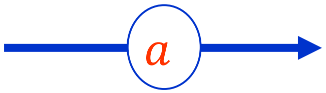
Ziffer:



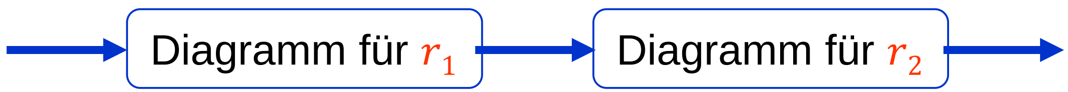
Kombinationen von Syntaxdiagrammen

ε

a



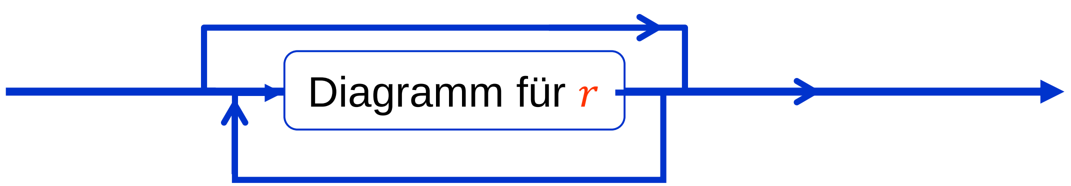
$r_1 \, r_2$



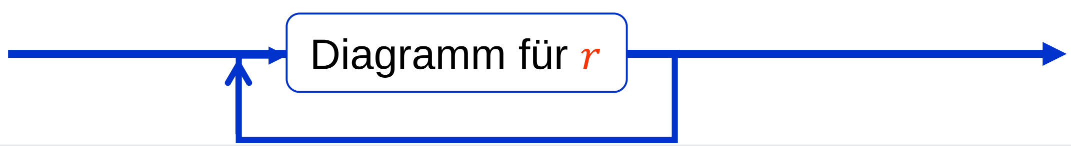
Keine Rekursion erlaubt!

$r_1 \mid r_2$
// $r_1 + r_2$

r^*

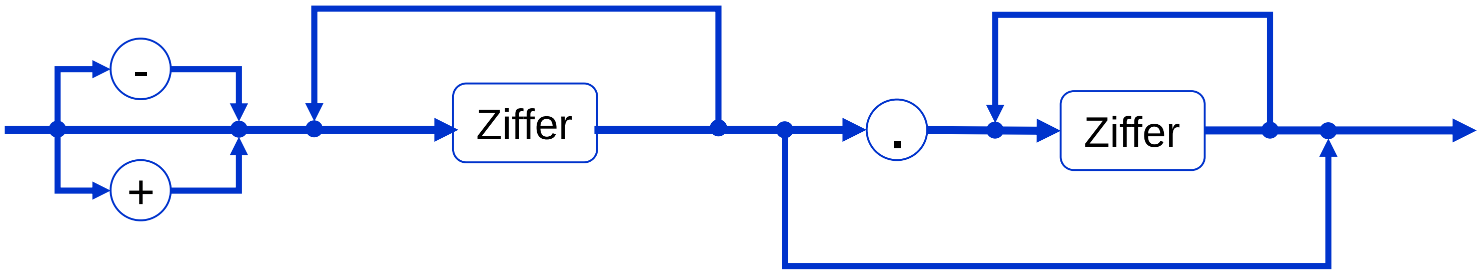


r^+



Regulärer Ausdruck

- Textuelle Beschreibung von Syntaxdiagrammen



  - Real = (+ | - | ε) Ziffer$^+$ (. Ziffer$^+$ | ε)
  - Ziffer = (0 | 1 | … | 9)

Reguläre Ausdrücke über Σ

- Konstante Ausdrücke
  - $\emptyset$,
  - ε,
  - a für jedes $a \in \Sigma$
- Sind e und f reguläre Ausdrücke, dann auch
  - $e + f$ // Alternative
  - $e\,f$ // Konkatenation
  - e^* // Kleene-Star
- Ist e regulärer Ausdruck, dann auch
  - (e) // Klammern erlaubt

Häufig „|" statt „+"

Beispiele für $\Sigma = \{0,1\}$: $0, \varepsilon, 0(10 + 01)^*1, 0(0 + 1)^*1, (0 + 1)^*0(0 + 1)^*, \ldots$

Beispiele für $\Sigma = \{a, \ldots, z\}$: $if, (bla)^*, \varepsilon, java(\varepsilon + script)(bla)^*, \ldots$

Beispiel für $\Sigma = \{0, \ldots, 9\}$:

$$0 + (1 + 2 + 3 + 4 + 5 + 6 + 7 + 8 + 9)(0 + 1 + 2 + 3 + 4 + 5 + 6 + 7 + 8 + 9)^*$$

Vereinbarung: * bindet am stärksten, + am schwächsten

Semantik regulärer Ausdrücke

- Zu jedem regulären Ausdruck e definiere eine Sprache $L(e)$:
  - $L(\emptyset) := \{\,\}$
  - $L(\varepsilon) := \{\varepsilon\}$
  - $L(a) := \{a\}$ für jedes $a \in \Sigma$
- und induktiv:
  - $L(e+f) := L(e) \cup L(f)$
  - $L(e\,f) := L(e) \circ L(f)$
  - $L(e^*) := L(e)^*$

- Beispiele:

$$L((0+1)^*\,0\,(1+(01)^*))$$
$$= (\{0\} \cup \{1\})^*\,\{0\}\,(\{1\} \cup \{01\}^*)$$
$$= \{0,1\}^*\,(\{01\} \cup \{0\}\{01\}^*)$$
$$= \{0,1\}^*\,(\{01\} \cup \{0\})$$

= alle Worte, die auf 0 oder auf 01 enden

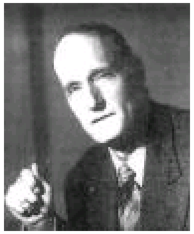
$$L(e+\emptyset) = L(e) \cup L(\emptyset)$$
$$= L(e) \cup \{\,\}$$
$$= L(e)$$

aber

$$L(e\,\emptyset) = L(e)\,L(\emptyset) = L(e)\{\,\} = \{\,\}$$
$$L(e\,\varepsilon) = L(e)\{\varepsilon\} = L(e)$$

Vorsicht: - ist kein regulärer Operator !

Gleichungen



Stephen Kleene

Definition: $r_1 \equiv r_2 :\Leftrightarrow L(r_1) = L(r_2)$

- Es gelten u.a.

$$\begin{gathered}
\emptyset + e \equiv e \\
e + f \equiv f + e \\
(e + f) + g \equiv e + (f + g) \\
\\
\varepsilon\, e \equiv e \equiv e\, \varepsilon \\
(e\, f)\, g \equiv e\, (f\, g)
\end{gathered}$$

$$\begin{gathered}
e(f + g) \equiv ef + eg \\
(e + f)g \equiv eg + fg \\
\\
\varepsilon^* \equiv \varepsilon \\
(e^*)^* \equiv e^* \\
(\varepsilon + e)^* \equiv e^* \\
(e^* f^*)^* \equiv (e + f)^* \\
(ef)^* e \equiv e\,(fe)^*
\end{gathered}$$

- Gilt die folgende Gleichung ?

$$\boxed{(ef + e)^* e \equiv e(fe + e)^*}$$

Herleitung

$$
\begin{aligned}
(e\,f + e)^*\,e &\equiv (e\,f + e\,\varepsilon)^* e \\
&\equiv \big(e\,(f + \varepsilon)\big)^* e \\
&\equiv e\,\big((f + \varepsilon)\,e\big)^* \\
&\equiv e\,(f\,e + \varepsilon\,e)^* \\
&\equiv e\,(f\,e + e)^*
\end{aligned}
$$

- Geht so eine Herleitung automatisch ?
- Gibt es einen Algorithmus zu entscheiden, ob $r_1 \equiv r_2$ gilt ?
  - Input: reguläre Ausdrücke r_1, r_2
  - Ausgabe: if ($r_1 \equiv r_2$) return true else return false … ?

Erweitert reguläre Ausdrücke (EBNF)

? und $^+$ für optionale und zu wiederholende Teile

- $e? := (e + \varepsilon)$
- $e^+ := ee^*$

[ ] für Teilmengen (Bereiche) des Alphabets:

- $[abc\ \ldots] := a|b|c|\ \ldots$
- $[a-z] := a|b|\ldots|z$ // zu lesen als a bis z
- $[a-z\text{äüöß}] := [a-z]\ |\ [\text{äüöß}]$

:= für Abkürzungen (Keine Rekursion erlaubt!)

- digit := $[0-9]$
- digits := digit*
- nat := $0\ |\ [1-9]$digits
- sign := $(+|-)?$

Beispiel: Gleitkommazahlen:

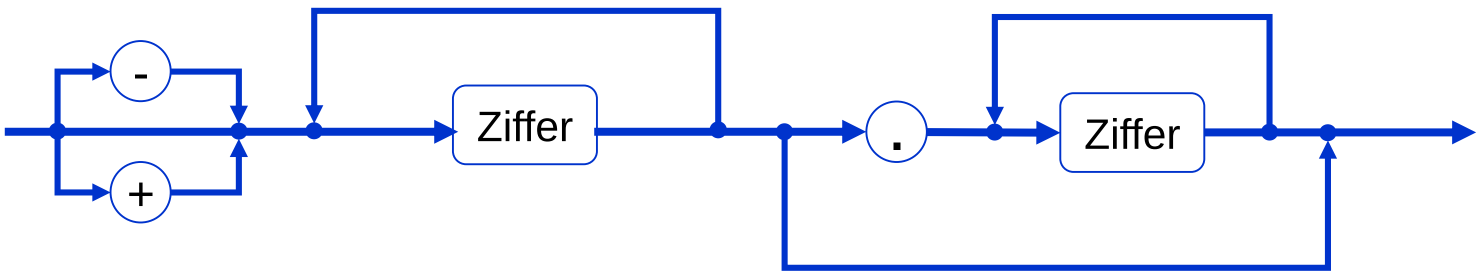
$[+\backslash-]?\ (0|[1-9]\text{digit}^*)\ (.[0-9]^+\ (E[+\backslash-]?\,\text{digit digit})?\ )?$ // „\-“ das Zeichen „-“, nicht

wobei // der Bereichsoperator „-“

$\text{digit} := [0-9]$

Beispiel: Syntaxdiagramm

- Real Zahlen
  - als Syntaxdiagramm



  - als regulärer Ausdruck:

$(+ \mid - \mid \varepsilon)\mathrm{Ziffer}^{+}(.\mathrm{Ziffer}^{+})?$ // „+“ als Zeichen, „|“ als Operator

wobei

$\mathrm{Ziffer} = (0 \mid 1 \mid \ldots \mid 9)$

Variationen und Zusätze

- statt + wird oft | benutzt
- [^ ...] Negation auf Zeichenmengen!
  - $[\hat{}aeiou] = [bcdfghj-np-tv-z]$
  - $[\hat{}x-z] = [a-w]$
- Vorgeschriebene Wiederholungen $\{n, m\}$
  - $e\{2{,}4\} = ee + eee + eeee$
  - $e\{3{,}*\} = eeee^*$ // $eee(e^*)$
- Escaping: Zeichen, verlieren Sonder-Bedeutung
  - [()]* eine Folge von Klammern
  - $[\backslash\mathrm{n}]$ neue Zeile
  - $\backslash **$ eine Folge von Sternen
  - $\backslash\backslash$ ein backslash

Programmiersprachen

- Teile von Programmiersprachen beschreibt man durch reguläre Ausdrücke
  - Ziffern
    - digit := $[0-9] \ = \ 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9$
  - Bezeichner
    - id := $[a-zA-Z\_\$][a-zA-Z\_\$0-9]^*$
  - HexZiffer
    - hex := <digit> | $[A-F]$
  - Unicode Zeichen
    - unic := \u <hex>
  - Gleitpunkt-Literale
    - dez := $[+\backslash-]?\ [0-9]^*\ (.[0-9]+)?\ (e\,[+\backslash-]?[0-9]+)?\ (d|f)?$

Token für reguläre Ausdrücke

- Reguläre Ausdrücke spezifizieren Token

  - $[a-zA-Z][a-zA-Z\_0-9]^*$ `return BEZEICHNER;`
  - $[+\backslash-]?\{dig\}+\backslash.?([eE][+\backslash-]?\{dig\}+)?$ `return NUM;`
  - `if|else|while` `return keyWord;`

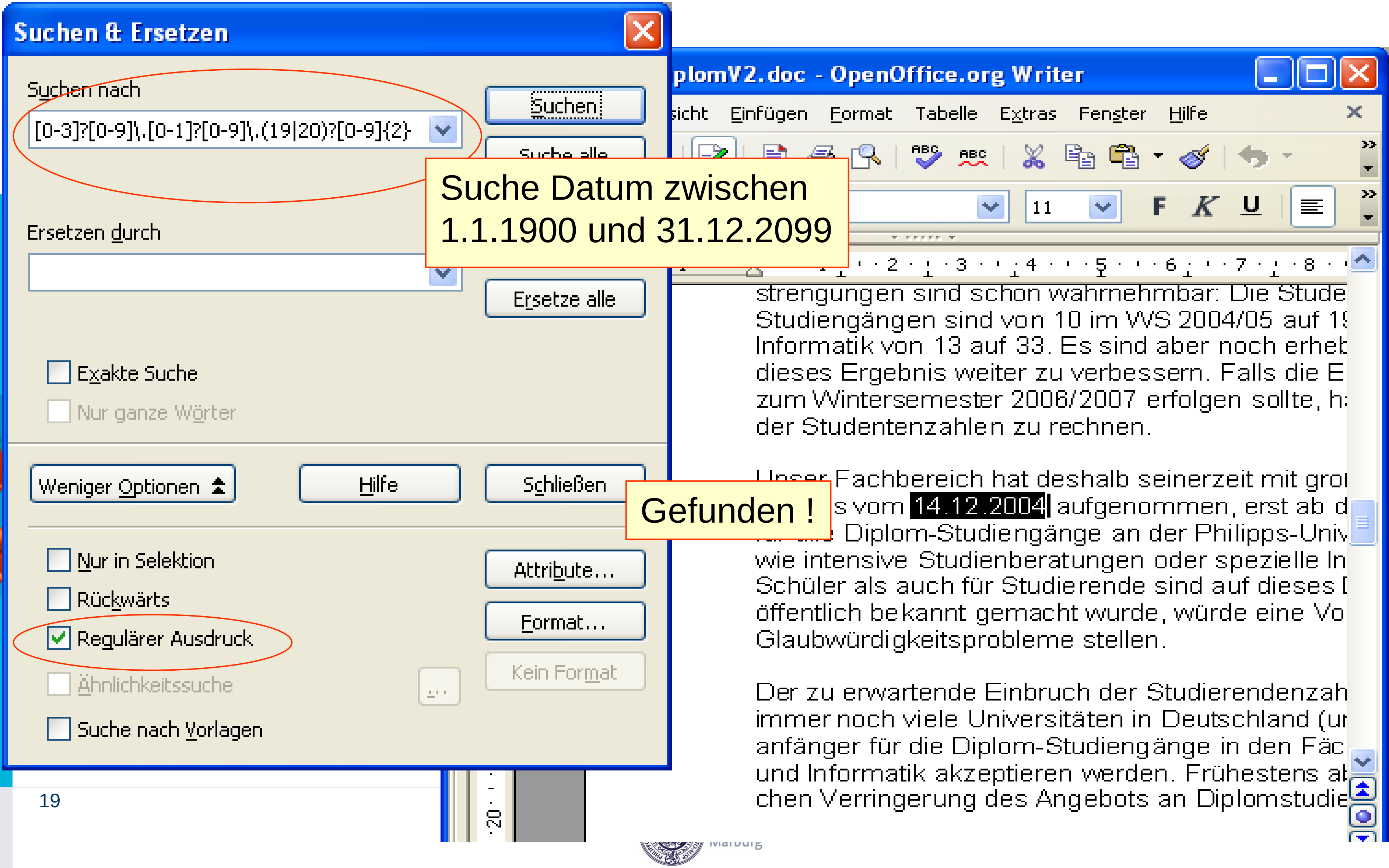
  - In (Open | Libre)Office steht „.“ für ein beliebiges Zeichen. Daher die escape-Sequenz \.

- Für jeden dieser Ausdrücke braucht man ein Programm, das diesen erkennt
  - oder ein Programm, das mehrere Ausdrücke erkennt und jeweils das entsprechende Token zurückgibt

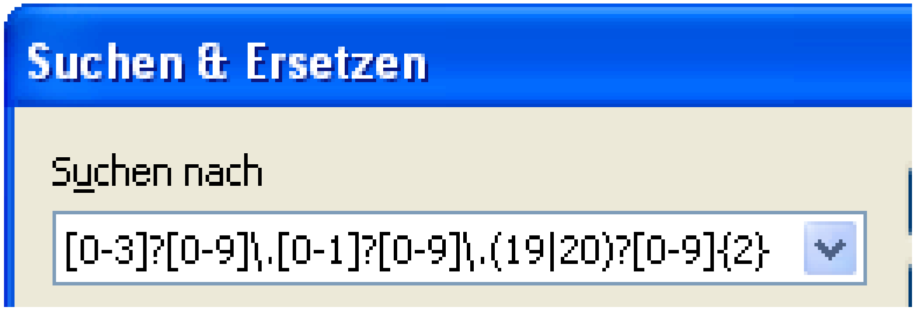
  - ➔ Scanner

- Scanner besitzt Spezifikation vieler regulärer Sprachen
  - Wenn er ein Wort einer Sprache erkennt, gibt er das entsprechende Token zurück

Suche in OpenOffice-Dokument



Der verwendete reguläre Ausdruck



Beispiele !

9.6.06	
21.07.2007	
5.22.09	
32.19.1999	

- $[0-3]?$
  - eine Ziffer zwischen 0 und 3 – optional
- $[0-9]$
  - eine beliebige Ziffer
- $\backslash.$
  - ein Punkt '.' Escapezeichen '\' notwendig
- $(19|20)?$
  - 19 oder 20 – optional
- $[0-9]\{2\}$
  - 2 beliebige Ziffern

Reguläre Ausdrücke in OpenOffice

OpenOffice.org Hilfe - OpenOffice.org Writer

Liste der regulären Ausdrücke

Zeichen	Wirkung/Einsatz
Beliebiges Zeichen	Steht für ein beliebiges einzelnes Zeichen, falls nich anders angegeben.
.	Steht für ein beliebiges einzelnes Zeichen außer einem Zeilen- oder einem Absatzumbruch. Beispielsweise liefert der Suchbegriff "Schmi.t" liefert sowohl "Schmitt" als auch "Schmidt".
*	Findet keines oder mehr der Zeichen vor dem "*". So liefert etwa der Suchbegriff "Ab*c" die Einträge "Ac", "Abc", "Abbc", "Abbbc" usw.
+	Findet ein oder mehr der Zeichen vor dem "+". Beispielsweise findet "AX.+4" zwar "AXx4", jedoch nicht "AX4". Es wird immer die längstmögliche Zeichenfolge gefunden, die dem Suchmuster in einem Absatz entspricht. Wenn der Absatz die Zeichenfolge "AX 4 AX4" enthält, wird der gesamte Ausdruck hervorgehoben.
?	Findet keines oder eines der Zeichen vor dem "?". Beispielsweise findet "Texts?" "Text" und "Texts" und "x(ab|c)?y" findet "xy", "xaby" oder "xcy".
[abc123]	Steht für eines der Zeichen in der Klammer.
[a-e]	Steht für ein beliebiges Zeichen im Buchstabenbereich a-e.
[a-eh-x]	Steht für ein beliebiges Zeichen im Buchstabenbereich a-e und h-x.
[^a-s]	Steht für ein beliebiges Zeichen außerhalb des Bereichs a-s.
\xXXXX	Steht für ein Zeichen auf Grundlage seines vierstelligen Hexadezimalcodes (XXXX). Der Code des Zeichens hängt von der jeweiligen Schrift ab. Die Codes können Sie unter **Einfügen - Sonderzeichen** einsehen.
dies|das	Findet die Begriffe, die vor oder hinter dem "|" auftreten. Beispielsweise findet "dies|das" sowohl "dies" als auch "das".
{2}	Gibt an, wie oft das Zeichen vor der öffnenden Klammer im Wort vorkommen muss. Zum Beispiel liefert der Suchbegriff "Man{2}" das Wort "Mann".
{1,2}	Gibt an, wie oft das Zeichen vor der öffnenden Klammer im Wort vorkommen darf. Zum Beispiel liefert der Suchbegriff "Man{1,2}" sowohl "Mann" als auch "man".
{1,}	Gibt an, wie oft das Zeichen vor der öffnenden Klammer im Wort mindestens vorkommen muss. Beispiel: Der Suchbegriff "Man{2}" findet "Mann", "Mannn" und "Mannnn".
( )	Die in der Klammer enthaltenen Zeichen gelten als Referenz. Auf die erste Referenz im aktuellen Ausdruck können Sie dann mit "\1" Bezug nehmen, auf die zweite mit "\2"

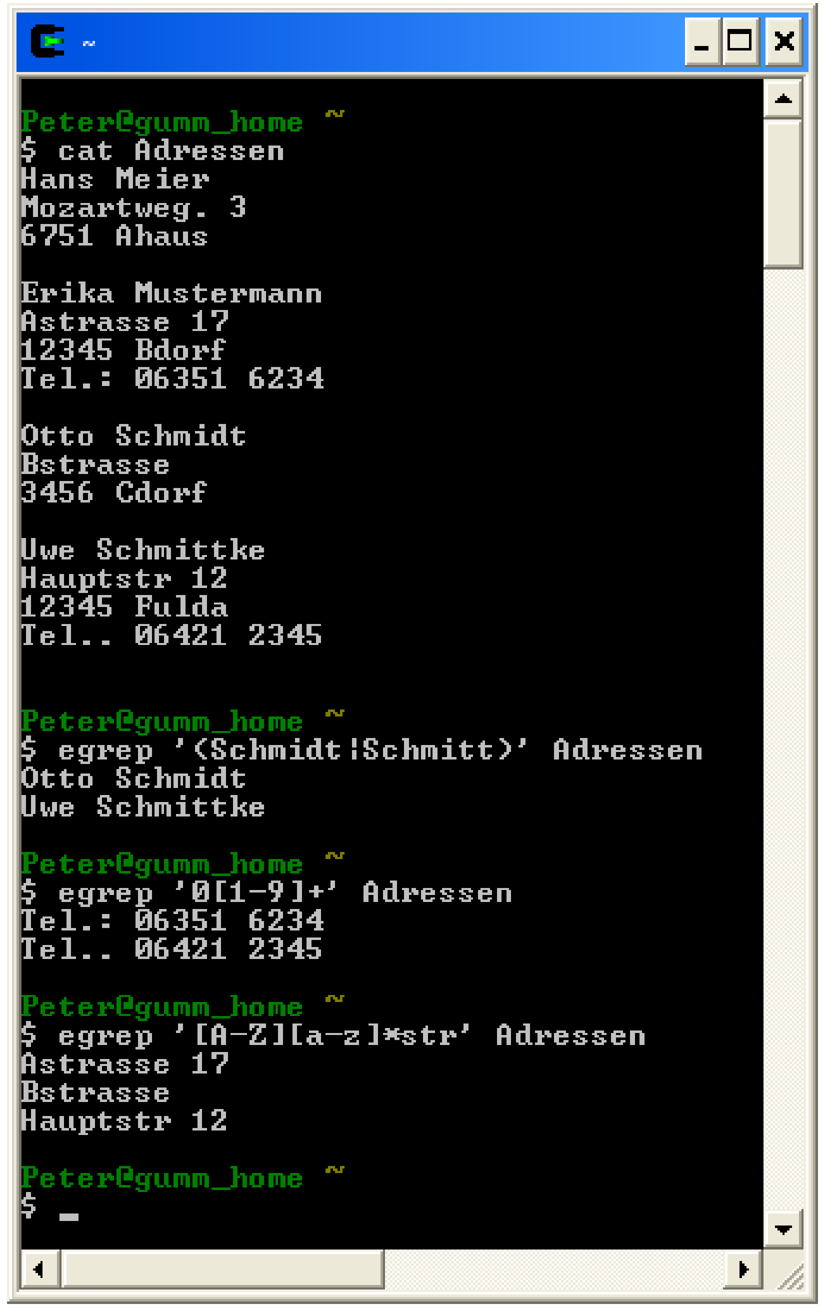
Reguläre Ausdrücke in Unix

Viele Unix-Kommandos akzeptieren reguläre Ausdrücke

- Suchkommandos
  - egrep (=grep –E)
- Editoren
  - sed
  - vi
  - emacs

Nicht verwechseln mit *File-Pattern*

- werden von shell ausgewertet



In Programmiersprachen

- Beispiel: Tool zum Parsen von Flugplänen
- Input:

Sun 14:30 Nice (NCE) 16:10 Brussels (BRU) easyJet U2 1643 Non-stop Airbus A320 (320) 1:40 Effective 2021-10-31 through 2022-03-20

Tue 10:05 Nice (NCE) 11:45 Brussels (BRU) easyJet U2 1645 Non-stop Airbus A320 (320) 1:40 Valid until 2021-08-31

Mon,Fri 10:55 Nice (NCE) 12:35 Brussels (BRU) easyJet U2 1645 Non-stop Airbus A320 (320) 1:40 Effective 2021-07-19 through 2021-09-03

Mon,Wed-Sun 12:55 Nice (NCE) 2 14:35 Brussels (BRU) Brussels Airlines SN 3618 Non-stop Airbus A320 (320) 1:40 Valid until 2021-07-17

Tue 12:55 Nice (NCE) 2 14:35 Brussels (BRU) Brussels Airlines SN 3618 Non-stop Airbus A320 (320) 1:40 Effective 2021-07-06 through 2021-10-26

Mon,Wed-Sun 12:55 Nice (NCE) 2 14:35 Brussels (BRU) Brussels Airlines SN 3618 Non-stop Airbus A320 (320) 1:40 Effective 2021-07-19 through 2021-10-30

Sun 13:45 Nice (NCE) 2 15:25 Brussels (BRU) Brussels Airlines SN 3618 Non-stop Airbus A319 (319) 1:40 Operates only on 2021-10-31

…

In Programmiersprachen

- Beispiel: Tool zum Parsen von Flugplänen
- Output:

```
flights on 2021-10-26
12:55 -> 14:35  NCE-BRU  - T - - - - -  SN3618 320 (2021-07-06, 2021-10-26)
12:55 -> 14:35  NCE-BRU  M - W T F S S  SN3618 320 (2021-07-19, 2021-10-30)
15:20 -> 17:00  NCE-BRU  M - W T F - -  U21645 319 (2021-09-06, 2021-10-29)
17:50 -> 19:30  NCE-BRU  - - - - - S -  SN3622 319 (2021-08-28, 2021-10-30)
17:50 -> 19:30  NCE-BRU  M T W T F - -  SN3622 320 (2021-08-31, 2021-10-29)
```

In Programmiersprachen

- z.B. in Python:

```
import re
OPDAYS_REGEX = re.compile('([a-zA-Z,-]+)\s*')
TIME_REGEX = re.compile('([0-9]{1,2}):([0-9]{1,2})\s*')
AIRPORT_REGEX = re.compile('.*?\(([A-Z]{3,3})\)\s*')
FLTNR_REGEX = re.compile(
    '''([a-zA-Z0-9 .'-]*?) ([A-Z0-9]{2,2} \d+)\s*''')
```

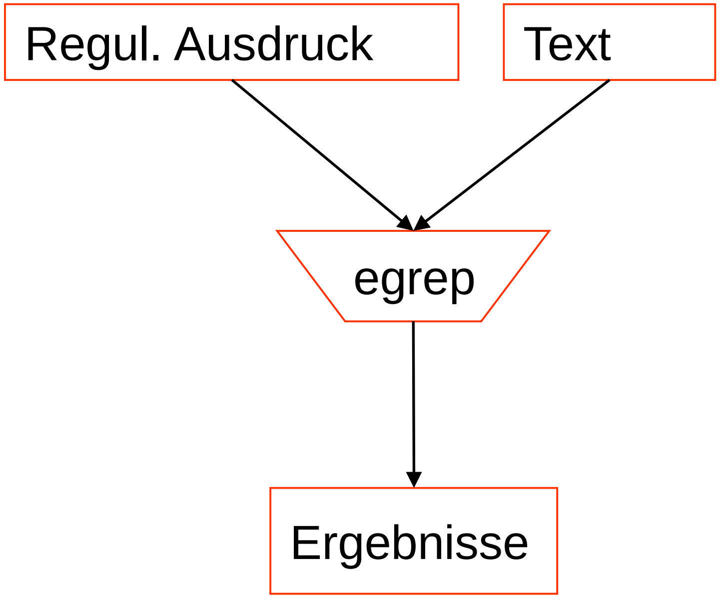
…

- ```
  def parse_time():
      m = TIME_REGEX.match(buffer.buf)
      buffer.advance(m.end())
      return time(*map(int, m.groups()))
  ```

…
  ```

Wie funktioniert Suche mit RE ?

- Wie mächtig sind reguläre Ausdrücke?
  - Was kann man ausdrücken, was nicht?
    - Nur „gültige“ Kalenderdaten?
      - 31.1.2006, 29.2.1996 aber nicht 30.2.04, 31.11.99
- Wie finde ich einen Teilstring, der auf einen regulären Ausdruck passt?
  - Algorithmus gesucht
  - Wie komplex ist so ein Algorithmus?
  - Effiziente Implementierung



Mit regulären Ausdrücken lösbar?

- Finde alle Worte
  – der Länge 5
  – die mit „(“ beginnen und mit „)“ enden
  – in denen mindesten 3 „f“ vorkommen

- Finde alle Worte
  – in denen mehr „A“-s als „B“-s vorkommen
  – in denen die Klammern „(„ und „)„ immer paarweise auftreten

- Wie kann man wissen, dass eine Frage mit regulären Ausdrücken nicht lösbar ist?

Mit regulären Ausdrücken lösbar?

- Finde alle Worte
  - der Länge 5
  - die mit „(“ beginnen und mit „)“ enden
  - in denen mindestens 3 „f“ vorkommen

lösbar

- Finde alle Worte
  - in denen mehr „A“-s als „B“-s vorkommen
  - in denen die Klammern „(„ und „)„ immer paarweise auftreten

nicht lösbar

- Wie kann man wissen, dass eine Frage mit regulären Ausdrücken nicht lösbar ist?

Zusammenfassung

- Reguläre Ausdrücke
  - beschreiben Wörter
    - in Programmiersprachen
    - in natürlichen Sprachen
    - entsprechen einfachen Syntaxdiagrammen
  - für Stringsuche verwendet
    - Grep, LibreOffice, etc.
  - Gleichungskalkül für reguläre Operatoren
    - Kleene Algebra
    - u.a. zur Vereinfachung