

# Reguläre Sprachen, Ausdrucksstärke

---

BC George (FH Bielefeld)

Unless otherwise noted, this work is licensed under CC BY-SA 4.0.

# Motivation

---

## Was muss ein Compiler wohl als erstes tun?

- Aufteilen des Inputs in  
Token

z.B. Schlüsselwörter, Zahlen  
(Konstanten), Operatoren

# Themen für heute

- Endliche Automaten
- Reguläre Ausdrücke

# Endliche Automaten

---

# Alphabete

**Def.:** Ein *Alphabet*  $\Sigma$  ist eine endliche, nicht-leere Menge von Symbolen. Die Symbole eines Alphabets heißen *Buchstaben*.

**Def.:** Ein *Wort*  $w$  über einem *Alphabet*  $\Sigma$  ist eine endliche Folge von Symbolen aus  $\Sigma$ .  $\epsilon$  ist das leere Wort. Die *Länge*  $|w|$  eines Wortes  $w$  ist die Anzahl von Buchstaben, die es enthält (Kardinalität).

**Def.:**  $\Sigma^k = \{w \text{ über } \Sigma \mid |w| = k\}$

$\Sigma^* = \bigcup_{i \in \mathbb{N}_0} \Sigma^i$  (die Kleene-Hülle von  $\Sigma$ )

$\Sigma^+ = \bigcup_{i \in \mathbb{N}} \Sigma^i$

$$\Sigma^0 = \{\epsilon\}$$

# Sprachen über Alphabete

**Def.:** Seien  $x = a_1 a_2 \dots a_n$  und  $y = b_1 b_2 \dots b_m$  Wörter. Wir nennen  $xy = x \circ y = a_1 \dots a_n b_1 \dots b_m$  die *Konkatenation* von  $x$  und  $y$ .  $= x \cdot y$

**Def.:** Eine Sprache  $L$  über einem Alphabet  $\Sigma$  ist eine Teilmenge von  $\Sigma^*$  :  $L \subseteq \Sigma^*$

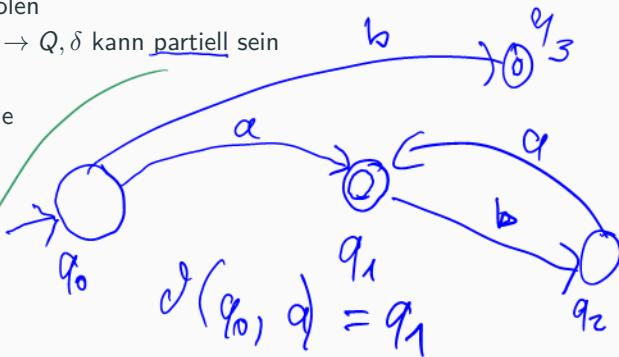
# Deterministische endliche Automaten

\*  $q_1$  :  $q_1$  ist Endzustand

**Def.:** Ein *deterministischer endlicher Automat* (DFA) ist ein 5-Tupel  $A = (Q, \Sigma, \delta, q_0, F)$  mit

- $Q$  : eine endliche Menge von Zuständen
- $\Sigma$  : ein Alphabet von Eingabesymbolen
- $\delta$  : die Übergangsfunktion  $(Q \times \Sigma) \rightarrow Q$ ,  $\delta$  kann partiell sein
- $q_0 \in Q$  : der Startzustand
- $F \subseteq Q$  : die Menge der Endzustände

	a	b
$\rightarrow q_0$	$q_1$	$q_3$
* $q_1$	—	$q_2$
$q_2$	$q_1$	—
* $q_3$	—	—





# Die Übergangsfunktion

alle Wörter in  $\Sigma^*$  sind endlich lang.

Def.: Wir definieren  $\delta^* : (Q \times \Sigma^*) \rightarrow Q$ : induktiv wie folgt:

- Basis:  $\delta^*(q, \epsilon) = q \quad \forall q \in Q$
- Induktion:  $\delta^*(q, a_1, \dots, a_n) = \delta(\delta^*(q, a_1, \dots, a_{n-1}), a_n)$

$$\delta^*(q_0, a b a) = q_1 \quad \text{s. o.}$$

Def.: Ein DFA akzeptiert ein Wort  $w \in \Sigma^*$  genau dann, wenn  $\delta^*(q_0, w) \in F$ .

Def.: Die Sprache eines DFA  $A$   $L(A)$  ist definiert durch:

$$L(A) = \{w \mid \delta^*(q_0, w) \in F\}$$

## Beispiel

Bsp. durch 3 teilbare Multipliken.

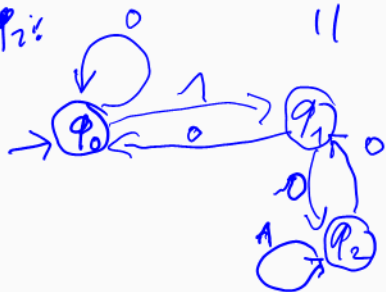
Zustände entsprechen Restklassen.

$q_0$ : bisher gelesene Zahl ergibt Rest 0 (Div. durch 3)

$q_1$ : 1

$q_2$ : 2

Schon gesehen  
0101 01



Eine 0 lesen:  
den bisherigen Wert mit 2 multiplizieren

Eine 1 lesen:  
den bisherigen Wert mit 2 multiplizieren  
+ 1

# Nichtdeterministische endliche Automaten

**Def.:** Ein *nichtdeterministischer endlicher Automat* (NFA) ist ein 5-Tupel  $A = (Q, \Sigma, \delta, q_0, F)$  mit

- $Q$ : eine endliche Menge von Zuständen
- $\Sigma$ : ein Alphabet von Eingabesymbolen
- $\delta$ : die Übergangsfunktion  $(Q \times \Sigma) \rightarrow \mathcal{P}(Q)$
- $q_0 \in Q$ : der Startzustand
- $F \subseteq Q$ : die Menge der Endzustände

*Potenzmenge von  $Q$ : Menge aller Teilmengen von  $Q$ .*

# Die Übergangsfunktion eines NFAs

**Def.:** Wir definieren  $\delta^* : (Q \times \Sigma) \rightarrow \mathcal{P}(Q)$  : induktiv wie folgt:

- Basis:  $\delta^*(q, \epsilon) = q \ \forall q \in Q$
- Induktion: Sei  $w \in \Sigma^*$ ,  $w = xa$ ,  $x \in \Sigma^*$ ,  $a \in \Sigma$  mit

$\delta^*(q, x) = \{p_1, \dots, p_k\}$ ,  $p_i \in Q$ , sei

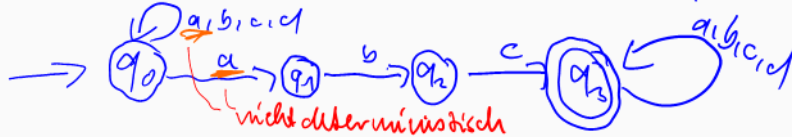
$$A = \bigcup_{i=1}^k \delta(p_i, a) = \{r_1, \dots, r_m\}, r_j \in Q.$$

Dann ist  $\delta^*(q, w) = \{r_1, \dots, r_m\}$ .

# Wozu NFAs im Compilerbau?

$\bar{Z} = \{a - \bar{a}\}$

Ist das Pattern  $abc$  enthalten?



Pattern Matching geht mit NFAs.

NFAs sind so nicht zu programmieren, aber:

**Satz:** Eine Sprache  $L$  wird von einem NFA akzeptiert  $\Leftrightarrow L$  wird von einem DFA akzeptiert.

# Konvertierung eines NFAs in einen DFA

Gegeben: Ein NFA  $A = (Q, \Sigma, \delta, q_0, F)$

Wir konstruieren einen DFA  $A' = (Q', \Sigma, \delta', q_0, F')$  wie folgt:

```
Q' = {{q0}}  
for each q in Q':  
  for each a ∈ Σ:  
    n = {p | δ(r, a) = p, r ∈ q}  
    Q' = Q' ∪ {n}  
    δ'(q, a) = n  
F' = {q ∈ Q' | q ∩ F ≠ ∅}  
{p, q} -> pq // Umbenennung
```

**Abbildung 1:** Konvertierung NFA in DFA

# Beispiel

$NFA$

	$\delta$	a	b
$\rightarrow q_0$		$\{q_0\}$	$\{q_1, q_2\}$
$q_1$		$\{q_2\}$	$\{q_1\}$
$*q_2$		$\{-\}$	$\{q_0, q_2\}$

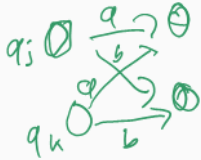
$\delta'$	a	b
$\rightarrow \{q_0\}$	$\{q_0\}$	$\{q_1, q_2\}$
$*\{q_1, q_2\}$	$\{q_2\}$	$\{q_0, q_1, q_2\}$
$*\{q_2\}$	-	$\{q_0, q_2\}$
$*\{q_0, q_2\}$	$\{q_0\}$	$\{q_0, q_1, q_2\}$
$*\{q_0, q_1, q_2\}$	$\{q_0, q_2\}$	$\{q_0, q_1, q_2\}$

max  $2^n$   
viele  
Zustände

# Minimierung eines DFAs

Ist ist der DFA  $A$  nicht vollständig, wird ein Fehlerzustand  $q_e$ , der kein Endzustand ist, hinzugefügt und in alle leeren Tabellenfelder eingetragen.

Dann wird eine Matrix generiert, die für alle Zustandspaare sagt, ob die beiden Zustände zu einem verschmelzen können.



```
for each (p,q) with  $p,q \in Q \times Q$ ,  $p \neq q$ :
  if ( $p \in F$  and  $q \notin F$ ) or ( $p \notin F$  and  $q \in F$ ):
     $D(p,q) = "-"$ 
  else
     $D(p,q) = \epsilon$ 

repeat
  for each (q,p)  $\in Q \times Q$  with  $p \neq q$ :
    for each  $a \in \Sigma$ :
      if  $D(p,q) = \epsilon$  and
          $D(\delta(p,a), \delta(q,a)) \neq \epsilon$ :
         $D(p,q) = a$ 
until there are no changes

for each entry in  $D$  with  $D(p,q) = \epsilon$ :
  combine  $p$  and  $q$  to one state
```

Abbildung 2: DFA Minimierung

- 1) Aus einem Pattern einen NFA entwickeln (s.o.)
- 2) Den NFA in einen DFA umwandeln
- 3) Den DFA minimieren



# Reguläre Ausdrücke

---

**Def.:** Seien  $L$  und  $M$  Sprachen.

- $L \cup M = \{w \mid w \in L \vee w \in M\}$
- $LM = L \cdot M = L \circ M = \{vw \mid v \in L \wedge w \in M\}$
- Die Kleene-Hülle einer Sprache:
  - Basis:  $L^0 = \{\epsilon\}$
  - Induktion:  $L^i = \{xw \mid x \in L^{i-1}, w \in L, i > 0\}$ ,  
$$L^* = \bigcup_{i \geq 0} L^i,$$
$$L^+ = \bigcup_{i > 0} L^i$$

# Reguläre Ausdrücke

**Def.:** Induktive Definition von regulären Ausdrücken (*regex*) und der von ihnen repräsentierten Sprache:

- Basis:

- $\epsilon$  und  $\emptyset$  sind reguläre Ausdrücke mit  $L(\epsilon) = \{\epsilon\}$ ,  $L(\emptyset) = \emptyset$
- Sei  $a$  ein Symbol  $\Rightarrow a$  ist ein regex mit  $L(a) = \{a\}$

- Induktion: Seien  $E$ ,  $F$  reguläre Ausdrücke. Dann gilt:

- $E + F$  ist ein regex und bezeichnet die Vereinigung  $L(E + F) = L(E) \cup L(F)$
- $EF$  ist ein regex und bezeichnet die Konkatenation  $L(EF) = L(E)L(F)$
- $E^*$  ist ein regex und bezeichnet die Kleene-Hülle  $L(E^*) = (L(E))^*$
- $(E)$  ist ein regex mit  $L((E)) = L(E)$

+ wird oft  
als | ge-  
schrieben,  
d.h.  
 $a+b = a|b$

Vorrangregeln der Operatoren für reguläre Ausdrücke: \*, Konkatenation, +

$$ab^* \neq (ab)^*$$

# Wichtige Identitäten

**Satz:** Sei  $A$  ein DFA  $\Rightarrow \exists$  regex  $R$  mit  $L(A) = L(R)$ .

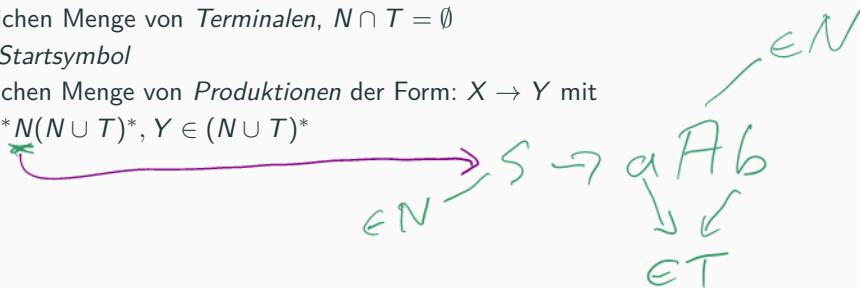
**Satz:** Sei  $E$  ein regex  $\Rightarrow \exists$  DFA  $A$  mit  $L(E) = L(A)$ .

# Formale Grammatiken

oft: Nichtterminale werden groß geschrieben, Terminale klein

**Def.:** Eine *formale Grammatik* ist ein 4-Tupel  $G = (N, T, P, S)$  aus

- $N$ : einer endlichen Menge von *Nichtterminalen*
- $T$ : einer endlichen Menge von *Terminalen*,  $N \cap T = \emptyset$
- $S \in N$ : dem *Startsymbol*
- $P$ : einer endlichen Menge von *Produktionen* der Form:  $X \rightarrow Y$  mit  $X \in (N \cup T)^* N (N \cup T)^*$ ,  $Y \in (N \cup T)^*$



# Ableitungen

**Def.:** Sei  $G = (N, T, P, S)$  eine Grammatik, sei  $\alpha A \beta$  eine Zeichenkette über  $(N \cup T)^*$  und sei  $A \rightarrow \gamma$  eine Produktion von  $G$ .

Wir sagen:  $\alpha A \beta \Rightarrow \alpha \gamma \beta$  ( $\alpha A \beta$  leitet  $\alpha \gamma \beta$  ab).

**Def.:** Wir definieren die Relation  $\Rightarrow^*$  induktiv wie folgt:

- Basis:  $\forall \alpha \in (N \cup T)^* \alpha \Rightarrow^* \alpha$  (Jede Zeichenkette leitet sich selbst ab.)
- Induktion: Wenn  $\alpha \Rightarrow^* \beta$  und  $\beta \Rightarrow \gamma$  dann  $\alpha \Rightarrow^* \gamma$

**Def.:** {Sei  $G = (N, T, P, S)$  eine formale Grammatik. Dann ist  $L(G) = \{w \in T^* \mid S \Rightarrow^* w\}$  die von  $G$  erzeugte Sprache.

**Def.:** Eine *reguläre (oder type-3-) Grammatik* ist eine formale Grammatik mit den folgenden Einschränkungen:

- Alle Produktionen sind entweder von der Form
  - $X \rightarrow aY$  mit  $X \in N, a \in T, Y \in N$  (*rechtsreguläre Grammatik*) oder
  - $X \rightarrow Ya$  mit  $X \in N, a \in T, Y \in N$  (*linksreguläre Grammatik*)
- $X \rightarrow \epsilon$  ist in beiden Fällen erlaubt.

**Satz:** Die von rechtsregulären Grammatiken erzeugten Sprachen sind genau die von linksregulären Grammatiken erzeugten Sprachen. Beide werden *reguläre* Sprachen genannt.

**Satz:** Die von regulären Ausdrücken beschriebenen Sprachen sind die regulären Sprachen.



# Das Pumping Lemma für reguläre Sprachen

**Satz:** Das Pumping Lemma für reguläre Sprachen:

Sei  $L$  eine <sup>unendliche</sup> reguläre Sprache.

$\Rightarrow \exists$  Konstante  $n \in \mathbb{N}$ : es gibt eine Wortlänge  $n$ , so dass für alle Wörter ab dieser Länge gilt:

$\forall \substack{w \in L \\ |w| \geq n} \exists x, y, z \in \Sigma^* \text{ mit } \underline{w = xyz}, y \neq \epsilon, |xy| \leq n :$

$\forall k \geq 0 \quad xy^kz \in L$

$\rightarrow$  kein  $\Leftrightarrow$ , d.h. für alle regulären Sprachen gilt das P.L., aber auch für andere Sprachen.

Aber: Wenn man beweisen kann, dass das P.L. nicht gilt, hat man bewiesen, dass eine Sprache nicht regulär ist.

Die Klasse der regulären Sprachen ist abgeschlossen unter

- Vereinigung
- Konkatenation
- Kleene-Stern
- Komplementbildung
- Durchschnitt

# Entscheidbarkeit für reguläre Sprachen

**Satz:** Es ist entscheidbar,

- ob eine gegebene reguläre Sprache leer ist
- ob  $w \in \Sigma^*$  in einer gegebenen regulären Sprache enthalten ist (Das “Wort-Problem”)
- ob zwei reguläre Sprachen äquivalent sind

# Grenzen der regulären Sprachen

Reguläre Sprachen sind von ihrer Struktur her einfach. Schon Sprachen, in denen etwas “gematcht” werden muss, lassen sich nicht mehr regulär beschreiben, weil z. B. die fixe Anzahl von Zuständen eines DFAs die Erkennung solcher Sprachen verhindert.

$a^n b^n$  ist nicht regulär.

Compiler müssen z.B. Klammern matchen.

# Wozu das Ganze?

Im Compilerbau werden reguläre Ausdrücke benutzt, um die Schlüsselwörter und weitere Symbole der zu erkennenden Sprache anzugeben. Daraus wird mit Hilfe eines Generators, der aus den regulären Ausdrücken DFAs (oder einen großen DFA) macht, der sog. Scanner oder Lexer genannt, generiert. Seine Aufgabe ist es, die Folge von Zeichen in der Quelldatei in eine Folge von sog. Token umzuwandeln. Z. B. wird so aus den Zeichen des Schlüsselwortes *while* im Programmtext das Token für *while* gemacht, das in der Syntaxanalyse weiterverarbeitet wird. Die Tokenfolge eines Programms ist ein Wort einer Sprache, die der Parser erkennt. Jedes vom Lexer erkannte Token ist dort also ein terminales Symbol.

# Ein Lexer ist mehr als ein DFA

Was ist zu beachten:

- Man braucht mindestens eine Liste von Paaren aus regulären Ausdrücken und Tokennamen.
- Neben den Schlüsselwörtern und Symbolen wie (,), \*, ... müssen auch Namen für Variablen, Funktionen, Klassen, Methoden, ... (sog. Identifier) erkannt werden
- Namen haben meist eine gewisse Struktur, die sich mit regulären Ausdrücken beschreiben lassen.
- Erlaubte Token sind in der Grammatik des Parsers beschrieben, d. h. für literale Namen, Strings, Zahlen liefert der Scanner zwei Werte:
  - z. B. <ID, "radius">, <Integerzahl, 558>
- Kommentare und Strings müssen richtig erkannt werden. (Schachtelungen)

Man kann natürlich auch einen Lexer selbst programmieren, d. h. die DFAs für die regulären Ausdrücke implementieren.

## Wrap-Up

---

- Definition und Aufgaben von Lexern
- DFAs und NFAs
- Reguläre Ausdrücke
- Reguläre Grammatiken
- Zusammenhänge zwischen diesen Mechanismen und Lexern, bzw. Lexergeneratoren





Unless otherwise noted, this work is licensed under CC BY-SA 4.0.