Pattern Matching

Definitionen:

1. *Alphabet*: endliche Menge von Zeichen, die *Buchstaben* genannt werden.
2. *Wörter*: Sei Σ Alphabet. Ein Σ–*Wort* ist endliche Folge von Buchstaben aus Σ:

   $$w = b_1 b_2 \cdots b_n \qquad \text{mit } b_i \in \Sigma$$

   ist Σ–Wort der Länge n

   Σ^*: Menge der Σ–Wörter
3. Das *leere Wort* bezeichnen wir mit ε.
4. *Konkatenation* von Wörtern:

   Seien $v = b_1 b_2 \cdots b_m \in \Sigma^*$ und $w = c_1 c_2 \cdots c_n \in \Sigma^*$.

   Wir definieren die Konkatenation vw von v und w als

   $$vw := b_1 b_2 \cdots b_m c_1 c_2 \cdots c_n$$
5. *Sprache*:

   Eine beliebige Teilmenge $\mathcal{L} \subseteq \Sigma^*$ heißt Σ–*Sprache*.

Beispiel: Sei $\Sigma = \{a, b\}$. Dann gilt

1. $\Sigma^* = \{\varepsilon, a, b, aa, ab, ba, bb, aaa, \cdots\}$
2. $\mathcal{L}_Q := \{w \in \Sigma^* \mid \exists v \in \Sigma^* : w = vv\}$
   ist eine Σ–Sprache

Syntax–Analyse (Chomsky–Hierarchie)

Aufgaben der *Syntax–Analyse* in der Informatik

1. Beschreibung von Σ–Sprachen
2. Bereitstellung von Algorithmen zum Erkennen von Σ–Wörtern

Mögliche Ansätze

1. Beschreibung durch *reguläre Ausdrücke*
   Erkennung über *endliche Automaten*
2. Beschreibung durch *kontext–freie* Grammatik
   Erkennung durch *Keller–Automaten*
3. Beschreibung durch *kontext–sensitive* Grammatik
   algorithmisch entscheidbar, exponentieller Aufwand
4. Beschreibung durch beliebige Grammatik
   unentscheidbar

Praktische Bedeutung in Informatik:

1. Reguläre Ausdrücke
   (a) Scanner: Aufteilung eines Programms in *Identifier*, *Schlüsselwörter*, *Literale*, *Kommentare*, etc.
   (b) Skript–Sprachen: *Tcl*, *Perl*, *Phyton*, $\cdots$
2. Kontext–freie Grammatik
   Parser: Strukturierung eines Programms in *Ausdrücke*, *Statements*, *Funktionen*, $\cdots$

Reguläre Ausdrücke

Gegeben: Alphabet Σ

Induktive Definition der

(a) Menge $\texttt{RegExp}$ der *regulären Ausdrücke* und der

(b) Sprache $\mathcal{L}(r)$ für $r \in \texttt{RegExp}$

1. $\emptyset \in \texttt{RegExp}$:

   $$\mathcal{L}(\emptyset) = \{\}$$

2. $\varepsilon \in \texttt{RegExp}$:

   $$\mathcal{L}(\varepsilon) = \{\varepsilon\}$$

3. $b \in \texttt{RegExp}$ für alle $b \in \Sigma$

   $$\mathcal{L}(b) = \{b\}$$

4. Konkatenation: $r_1, r_2 \in \texttt{RegExp} \ \Rightarrow \ r_1 r_2 \in \texttt{RegExp}$

   $$\mathcal{L}(r_1 r_2) = \{vw \in \Sigma^* \mid v \in \mathcal{L}(r_1) \wedge w \in \mathcal{L}(r_2)\}$$

5. Alternative: $r_1, r_2 \in \texttt{RegExp} \ \Rightarrow \ (r_1|r_2) \in \texttt{RegExp}$

   $$\mathcal{L}\big((r_1|r_2)\big) = \mathcal{L}(r_1) \cup \mathcal{L}(r_2)$$

6. Abschluß: $r \in \texttt{RegExp} \ \Rightarrow \ (r)^* \in \texttt{RegExp}$

   $$\mathcal{L}\big((r)^*\big) = \big\{\varepsilon\big\} \cup \big\{w_1 \cdots w_n \mid w_i \in \mathcal{L}(r)\big\}$$

Vereinfachung von regulären Ausdrücken

Bindungs–Regeln

1. Abschluß bindet stärker als Konkatenation
2. Konkatenation bindet stärker als Alternative

Dann können Klammer weggelassen werden:

1. ab^* steht für $a(b)^*$
2. $a|b|c$ steht für $((a|b)|c)$

Alternative Schreibweise:

$r_1 + r_2$ statt $r_1|r_2$

Definition: $r \simeq s \Leftrightarrow \mathcal{L}(r) = \mathcal{L}(s)$

Eigenschaften regulärer Ausdrücke

1. $\emptyset + r \simeq r$, $\varepsilon r \simeq r\varepsilon = r$, $\emptyset^* = \varepsilon$, $\emptyset r = r\emptyset = \emptyset$
2. $r + s \simeq s + r$, $\varepsilon * = \varepsilon$
3. $r + r \simeq r$, $(r^*)^* \simeq r^*$
4. $(r + s) + t \simeq r + (s + t)$, $(rs)t \simeq r(st)$
5. $(r + s)t \simeq rt + st$, $t(r + s) \simeq tr + ts$
6. $(\varepsilon + r)^* \simeq r^*$, $rr^* = r^*r$
7. $(r + s)^* \simeq (r^*s^*)^*$
8. $(rs)^* \simeq \varepsilon + rs(rs)^*$, $(rs)^*r \simeq r(sr)^*$

Reguläre Ausdrücke: Erweiterungen & Beispiele

Abkürzungen: Sei $\Sigma = \{a, b, c, \cdots, x, y, z,\ 0, \cdots, 9\}$

`[abc]`	$\widehat{=}$	$a\|b\|c$
`[0-9]`	$\widehat{=}$	$0\|1\|2\|3\|4\|5\|6\|7\|8\|9$
`[^0-9]`	$\widehat{=}$	$a\|b\|c\|\cdots\|x\|y\|z$
`.`	$\widehat{=}$	$a\|b\|c\|\cdots\|x\|y\|z\|0\|\cdots\|9$
$r+$	$\widehat{=}$	$rr*$
$r?$	$\widehat{=}$	$\varepsilon\|r$

Beispiele

1. `[0-9]+`

   nicht–leere Folge von Ziffern, z. B. “01”

2. `0|[1-9][0-9]*`

   Zahl im Dezimal–System

3. `http://[^/]+/[^␣]+`

   (a) Wörtlich: “`http://`”

   (b) nicht–leere Folge von Buchstaben, die keinen Slash “/” enthält

   (c) Wörtlich: Slash “/”

   (d) nicht–leere Folge von Buchstaben, die kein Blank “␣” enthält

Endliche Automaten (Finite State Machines)

Definition: Das 5–Tupel

$$\langle \Sigma, Z, A, s_0, \texttt{next} \rangle$$

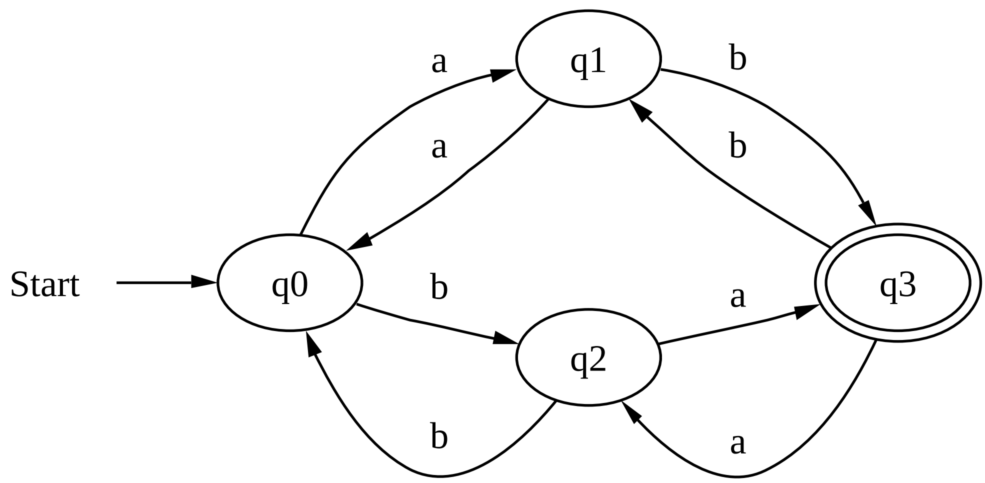
ist ein deterministischer endlicher Automat falls

1. Σ: Eingabe–Alphabet, endliche Menge
2. Z: Menge der Zustände, endlich
3. A: Menge der *akzeptierenden Zustände*, $A \subseteq Z$
4. s_0: Start–Zustand, $s_0 \in Z$
5. $\texttt{next} : Z \times \Sigma \rightarrow Z$
   heißt *Zustands–Übergangs–Funktion*

Graphische Darstellung:

1. Zustände: Kreis um Namen des Zustands
2. Akzeptierende Zustände: doppelte Kreise
3. Zustands–Übergangs–Funktion:
   Gilt $s_2 = \texttt{next}(s_1, a)$, so verbinden wir s_1 und s_2 durch Pfeil, der mit a beschriftet ist.
4. Start–Zustand: steht in Diagramm entweder ganz links oder ganz oben, Pfeil zeigt auf Start–Zustand

Ein Beispiel



Sei $F_{ug} := \langle \Sigma, Z, A, s_0, \mathtt{next} \rangle$ mit

1. $\Sigma = \{a, b\}$.
2. $Z = \{q_0, q_1, q_2, q_3\}$.
3. $A = \{q_3\}$.
4. $s_0 = q_0$.
5. $\mathtt{next}(q_0, a) = q_1$, $\mathtt{next}(q_0, b) = q_2$.
6. $\mathtt{next}(q_1, a) = q_0$, $\mathtt{next}(q_1, b) = q_3$.
7. $\mathtt{next}(q_2, a) = q_3$, $\mathtt{next}(q_2, b) = q_0$.
8. $\mathtt{next}(q_3, a) = q_2$, $\mathtt{next}(q_3, b) = q_1$.

Berechnung eines endl. Automaten

Gegeben: Endlicher Automat

$$F = \langle \Sigma, Z, A, s_0, \mathtt{next} \rangle$$

Induktive Definition der Berechnung eines endlichen Automaten:

1. $q_1 \stackrel{c}{\rightarrow} q_2$ Berechnung mit Label $c \in \Sigma$ falls
   $q_2 = \mathtt{next}(q_1, c)$

2. $q_0 \stackrel{x_1}{\rightarrow} \cdots \stackrel{x_n}{\rightarrow} q_n \stackrel{x_{n+1}}{\rightarrow} \cdots \stackrel{x_m}{\rightarrow} q_m$
   Berechnung mit Label vw falls
   (a) $q_0 \stackrel{x_1}{\rightarrow} \cdots \stackrel{x_n}{\rightarrow} q_n$ Berechnung mit Label v
   (b) $q_n \stackrel{x_{n+1}}{\rightarrow} \cdots \stackrel{x_m}{\rightarrow} q_m$ Berechnung mit Label w ist.

Definition der *akzeptierten Sprache*:

Ein Wort $w \in \Sigma^*$ ist genau dann in der akzeptierten Sprache $\mathcal{L}(F)$, wenn es eine Berechnung

$$s_0 \stackrel{x_0}{\rightarrow} \cdots \stackrel{x_n}{\rightarrow} q$$

mit Label w gibt und $q \in A$ liegt.

Beachte, dass Berechnung im Start–Zustand s_0 startet!

Beispiel: Betrachte letzten Automaten:

$$q_0 \stackrel{a}{\rightarrow} q_1 \stackrel{a}{\rightarrow} q_0 \stackrel{b}{\rightarrow} q_2 \stackrel{a}{\rightarrow} q_3$$

Berechnung mit Label $aaba$, also $aaba \in \mathcal{L}(F)$.

Arbeitsweise eines endlichen Automaten

Gegeben:

1. Endlicher Automat $F = \langle \Sigma, Z, A, s_0, \texttt{next} \rangle$
2. Wort $w \in \Sigma^*$

Frage: Wird $w = w[1] \cdots w[n]$ von F *akzeptiert*?

1. Beginne im Start–Zustand s_0:

   ```
   q = s0
   ```

   q is aktueller Zustand
2. Setze `idx = 1`
3. Berechne Folge–Zustand

   ```
   q = next(q, w[idx])
   ```
4. Inkrementiere `idx`.
5. Falls `idx` $\leq n$: Gehe zu 3.
6. Ist `q` $\in A$:
   (a) Ja: F hat w akzeptiert
   (b) Nein: F hat w nicht akzeptiert

C–Implementierung einer FSM

```
typedef enum { Q0, Q1, Q2, Q3 } State;

bool accept(const char* w)
{
    State q = Q0;
    while (*w != 0) {
        switch (q) {
        case Q0: {
            switch (*w) {
                case 'a': {
                    q = Q1;
                    break;
                }
                case 'b': {
                    q = Q2;
                    break;
                }
            }
            break;
        }
        case Q1: { ... }
        case Q2: { ... }
        case Q3: { ... }
        ++w;
    }
    if (q == Q3)
        return true;
    return false;
}
```

Erweiterung der Zustands–Übergangs–Funktion

Wir erweitern die Zustands–Übergangs–Funktion

$$\texttt{next} : Z \times \Sigma \to Z$$

zu einer Funktion

$$\texttt{next}^* : Z \times \Sigma^* \to Z$$

Definition von $\texttt{next}^*(w)$ induktiv über Länge von w

1. $\texttt{next}^*(q, \varepsilon) = q$
2. $\texttt{next}^*(q, bw) = \texttt{next}^*\big(\texttt{next}(q, b), w\big)$

Bemerkung: Endlicher Automat

$$F = \langle \Sigma, Z, A, s_0, \texttt{next} \rangle$$

akzeptiert $w \in \Sigma^*$ g.d.w.

$$\texttt{next}^*(s_0, w) \in A$$

Bemerkung: *akzeptierte Sprache*

$$\mathcal{L}(F) = \{w \in \Sigma^* \mid \texttt{next}^*(s_o, w) \in A\}$$

Beispiel von Folie 6:

Bezeichne $\texttt{nr}(x, w)$ Anzahl der Buchstaben x, die in w auftreten. Dann gilt:

$$\mathcal{L}(F_{ug}) = \big\{w \mid \texttt{nr}(a, w) \,\%\, 2 = 1 \land \texttt{nr}(b, w) \,\%\, 2 = 1\big\}$$

Nicht–deterministische Endliche Automaten

Definition: Das 6–Tupel

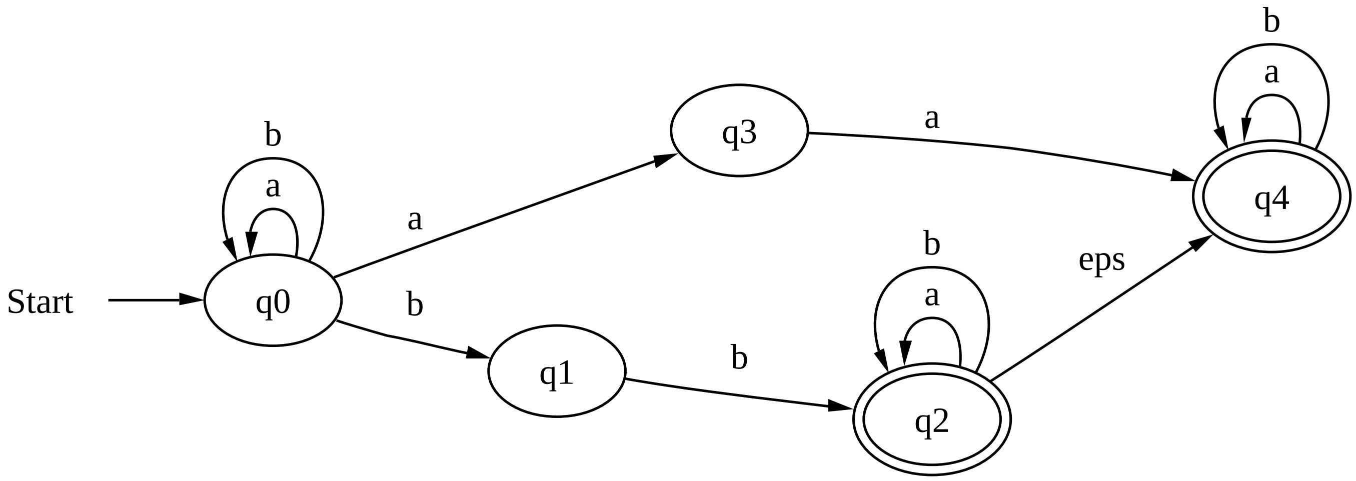
$$\langle \Sigma, Z, A, s_0, \texttt{next}, \texttt{eps} \rangle$$

ist nicht–deterministischer endl. Automat mit ε Übergängen falls

1. Σ: Eingabe–Alphabet, endliche Menge
2. Z: Menge der Zustände, endlich
3. A: Menge der *akzeptierenden Zustände*, $A \subseteq Z$
4. s_0: Start–Zustand, $s_0 \in Z$
5. $\texttt{next} : Z \times \Sigma \rightarrow 2^Z$

   heißt nicht–determ. *Zustands–Übergangs–Funktion*
6. $\texttt{eps} : Z \rightarrow 2^Z$

   heißt *Epsilon–Übergangs–Funktion*



Akzeptierte Sprache

Gegeben: Nicht–deterministischer endl. Automat

$$F = \langle \Sigma, Z, A, s_0, \mathtt{next}, \mathtt{eps} \rangle$$

Induktive Definition der Berechnung eines endlichen Automaten:

1. $q_1 \stackrel{\varepsilon}{\longrightarrow} q_2$ Berechnung mit Label ε falls $q_2 \in \mathtt{eps}(q_1)$

2. $q_1 \stackrel{c}{\longrightarrow} q_2$ Berechnung mit Label $c \in \Sigma$ falls $q_2 \in \mathtt{next}(q_1, c)$

3. $q_0 \stackrel{x_1}{\longrightarrow} \cdots \stackrel{x_n}{\longrightarrow} q_n \stackrel{x_{n+1}}{\longrightarrow} \cdots \stackrel{x_m}{\longrightarrow} q_m$

   Berechnung mit Label vw falls

   (a) $q_0 \stackrel{x_1}{\longrightarrow} \cdots \stackrel{x_n}{\longrightarrow} q_n$ Berechnung mit Label v

   (b) $q_n \stackrel{x_{n+1}}{\longrightarrow} \cdots \stackrel{x_m}{\longrightarrow} q_m$ Berechnung mit Label w ist.

Definition der *akzeptierten Sprache*:

Ein Wort $w \in \Sigma^*$ ist genau dann in der akzeptierten Sprache $\mathcal{L}(F)$, wenn es eine Berechnung

$$s_0 \stackrel{x_0}{\longrightarrow} \cdots \stackrel{x_n}{\longrightarrow} q$$

mit Label w gibt, so dass $q \in F$ ist.

Beachte, dass Berechnung im Start–Zustand s_0 startet!

Beispiel: Betrachte letzten Automaten:

$$q_0 \stackrel{a}{\longrightarrow} q_0 \stackrel{b}{\longrightarrow} q_1 \stackrel{b}{\longrightarrow} q_2 \stackrel{\varepsilon}{\longrightarrow} q_4 \stackrel{a}{\longrightarrow} q_4$$

Berechnung mit Label $abba$, also $abba \in \mathcal{L}(F)$.

Berechnung der akzeptierten Sprache

Wir erweitern die Funktion `next` und `eps` zu Funktionen

$$\texttt{Next} : 2^Z \times \Sigma \to 2^Z \quad \text{und} \quad \texttt{Eps} : 2^Z \to 2^Z$$

$$\texttt{Next}(Q, b) \;:=\; \big\{ z \in Z \mid \exists q \in Q : z \in \texttt{next}(q, b) \big\}$$

$$\texttt{Eps}(Q) \;:=\; Q \cup \big\{ z \in Z \mid \exists q \in Q : z \in \texttt{eps}(q) \big\}$$

Beispiel:

1. $\texttt{Next}(\{q_0\}, a) = \{q_0, q_3\}$
2. $\texttt{Next}(\{q_0, q_3\}, a) = \{q_0, q_3, q_4\}$
3. $\texttt{Eps}(\{q_0\}) = \{q_0\}$
4. $\texttt{Eps}(\{q_2\}) = \{q_2, q_4\}$

Wir erweitern die Funktion `Next` induktiv auf Worte w zu einer Funktion

$$\texttt{Next}^* : 2^Z \times \Sigma^* \to 2^Z$$

1. $\texttt{Next}^*(Q, \varepsilon) \;:=\; \texttt{Eps}(Q)$
2. $\texttt{Next}^*(Q, bw) \;:=\; \texttt{Next}^*\Big(\texttt{Next}\big(\texttt{Eps}(Q),\ b\big),\ w \Big)$

Beispiel:

1. $\texttt{Next}^*(\{q_0\}, aa) = \{q_0, q_3, q_4\}$
2. $\texttt{Next}^*(\{q_0\}, abba) = \{q_0, q_2, q_3, q_4\}$

Satz: Ist F nicht–determ. Automat, so gilt:

$$\mathcal{L}(F) \;=\; \big\{ w \in \Sigma^* \mid \texttt{Next}^*(\{s_0\}, w) \cap F \neq \emptyset \big\}$$

Potenz–Mengen–Konstruktion

Gegeben: Nicht–deterministische FSM

$$F_{nd} = \langle \Sigma, Z, A, s_0, \texttt{next}, \texttt{eps} \rangle$$

Gesucht: Deterministische FSM

$$F_{det} = \langle \Sigma, \mathcal{Z}, \mathcal{A}, S_0, \mathcal{NS} \rangle$$

mit $\mathcal{L}(F_{det}) = \mathcal{L}(F_{nd})$

Definition:

1. $\mathcal{Z} := 2^Z$
2. $\mathcal{A} := \big\{ M \in 2^Z \mid M \cap F \neq \emptyset \}$
3. $S_0 := \{s_0\}$
4. $\mathcal{NS}(Q) := \texttt{Eps}\Big(\texttt{Next}\big(\texttt{Eps}(Q), b \big) \Big)$

Zahl der Zustände: $|\mathcal{Z}| = 2^{|Z|}$: sehr groß!

Satz: $\mathcal{L}(F_{det}) = \mathcal{L}(F_{nd})$

Beobachtung: Nicht–deterministische Automaten sind nicht mächtiger als deterministische Automaten.

Übersetzung von regulären Ausdrücken in nicht–deterministische FSMs

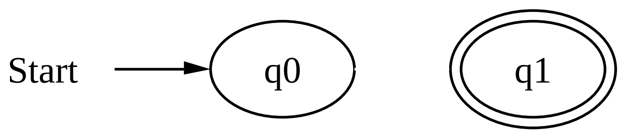
Wir definieren für jeden regulären Ausdruck r eine nicht–deterministische FSM durch Induktion über r

Konstruktions–Invarianten:

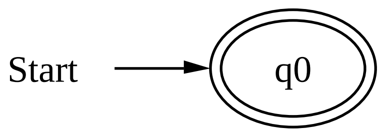
1. Menge der akzeptierenden Zustände ein–elementig
2. $|\texttt{eps}(q)| \leq 2$
3. $|\bigcup_{b \in \Sigma} \texttt{next}(q,b)| \leq 1$
4. $\texttt{eps}(q) = \emptyset \;\vee\; \forall b \in \Sigma : \texttt{next}(q,b) = \emptyset$

Induktive Definition:

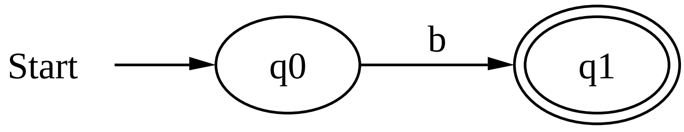
1. $r = \emptyset$:



2. $r = \varepsilon$:



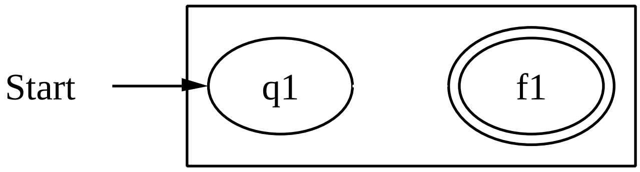
3. $r = b$ mit $b \in \Sigma$:



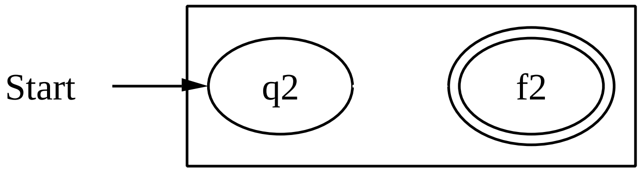
$\texttt{RegExp} \mapsto$ FSM (Fortsetzung)

Nach IV seien FSMs für r_1 und r_2 wie folgt gegeben:

1. r_1:

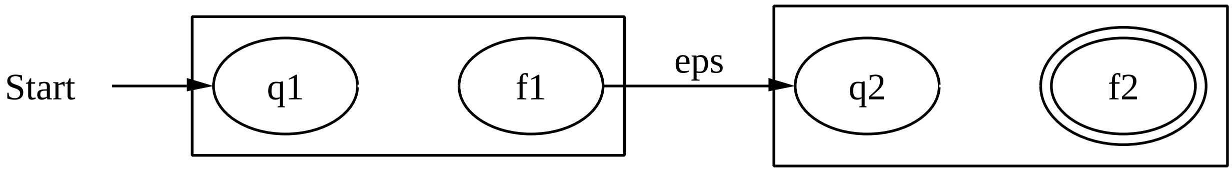


2. r_2:

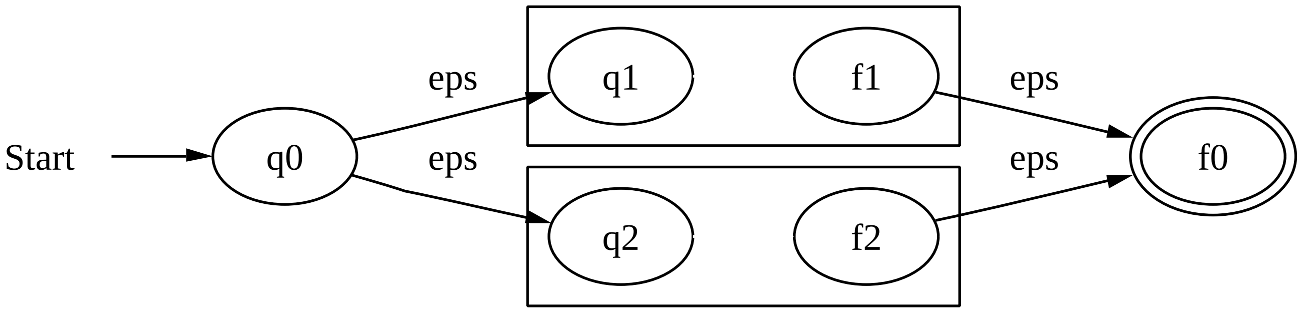


Fortsetzung der induktiven Definition:

4. $r = r_1 r_2$

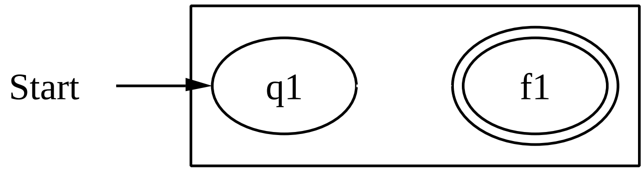


5. $r = r_1 + r_2$



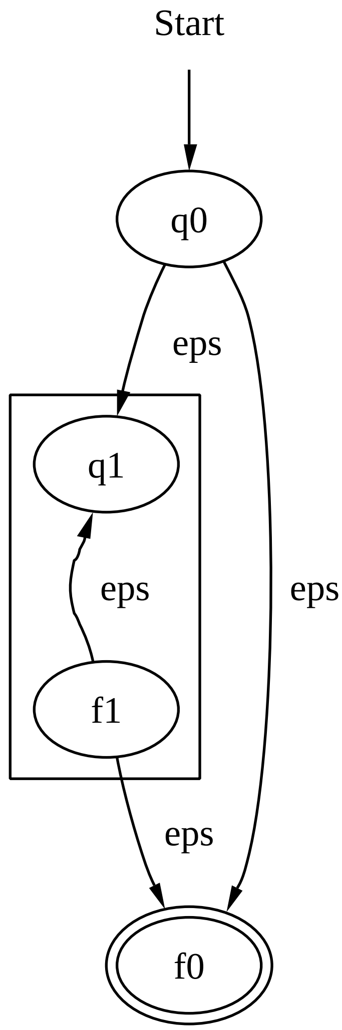
Konstruktion der FSM für r_1^*

Nach IV sei FSM für r_1 gegeben durch

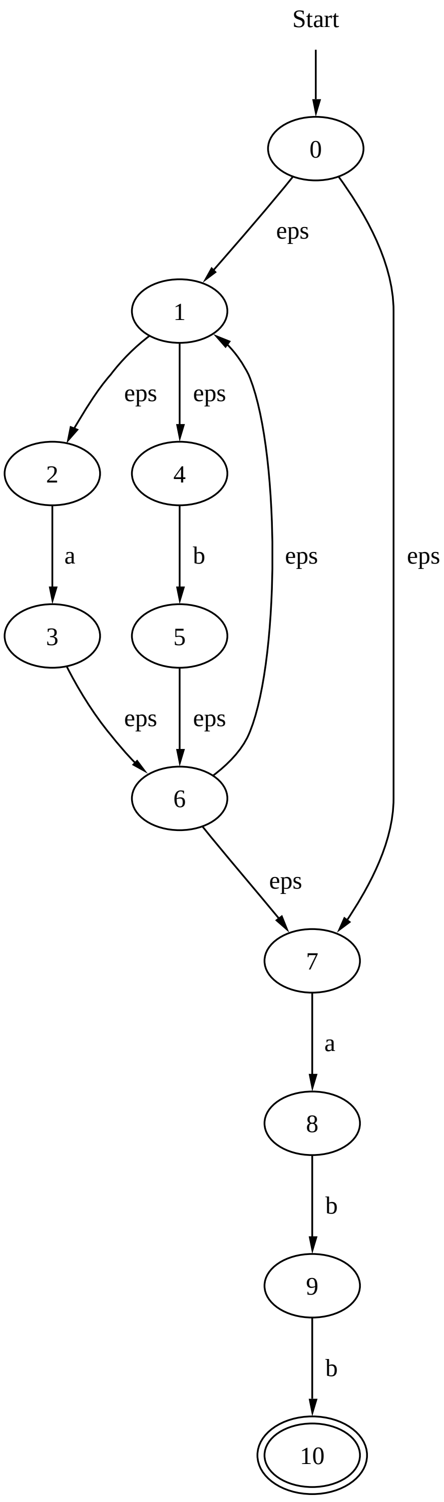


Fortsetzung der induktiven Definition

6. $r = r_1^*$



Beispiel: $(a|b)^*abb$



Simulation nicht–deterministischer FSMs

Problem: Größe Potenz–Mengen–Konstruktion: $2^{|Z|}$

Lösung: *Simulation* der Potenz–Mengen–Konstruktion

Repräsentation einer FSM

1. Anzahl der Zustände: `numberStates`
2. Codierung der Zustände als Zahlen:

   $$0, \cdots, \texttt{numberStates} - 1$$

3. Feld von Symbolen: `symbol[numberStates]`
4. Feld von Folge–Zuständen: `next1[numberStates]`
5. Feld von Folge–Zuständen: `next2[numberStates]`

Bedeutung dieser Felder:

1. $j \in \texttt{next}(i, c) \rightarrow \texttt{symbol}[i] = c$
2. $\big(\forall c \in \Sigma : \texttt{next}(i, c) = \emptyset\big) \rightarrow \texttt{symbol}[i] = 0$
3. $j \in \texttt{next}(i, c) \rightarrow \texttt{next1}[i] = j$
4. $j \in \texttt{eps}(i) \rightarrow \texttt{next1}[i] = j \vee \texttt{next2}[i] = j$

C–Daten–Struktur

```
typedef struct {
    unsigned  numberStates; // number of states
    char*     symbol;       // array of characters
    unsigned* next1;        // possible next state
    unsigned* next2;        // possible next state
} FSM;
```

Respresentation der Beispiel–FSM

```
fsm->numberStates = 11;

// Numerierung der Zustaende
//    0  1   2   3   4   5  6   7    8    9   10

fsm->symbol =
    { 0, 0, 'a', 0, 'b', 0, 0, 'a', 'b', 'b',  0 };

fsm->next1 =
    { 1, 2,  3,  6,  5,  6, 7,  8,   9,  10,  10 };

fsm->next2 =
    { 7, 4,  3,  6,  5,  6, 1,  8,   9,  10,  10 };
```

Berechnung des Epsilon–Abschluß

$$\texttt{epsClose} : 2^Z \to 2^Z$$

$\texttt{epsClose}(Q)$: Menge der Zustände, die von Zuständen in Q durch Epsilon–Übergange erreicht werden können.

Definition von $\texttt{epsClose}(Q)$ iterativ:

1. Für alle $Q \subseteq Z$ gilt:

   $$Q \subseteq \texttt{epsClose}(Q)$$

2. Für alle $Q \subseteq Z$ und alle $q \in Z$ gilt:

   $$q \in \texttt{epsClose}(Q) \;\Rightarrow\; \texttt{eps}(q) \subseteq \texttt{epsClose}(Q)$$

Berechnung von `epsClose`(Q)

Repräsentation von Zustands–Mengen $Q \subseteq Z$ durch Felder:

1. Für die Menge der Zustände gilt:

   $$Z = \{q_0, q_1, \cdots, q_n\}$$

   mit $n + 1 =$ `numberStates`.

2. $Q \subseteq Z$ wird dargestellt durch

   `bool state[numberStates]`

   Dabei gilt

   $\texttt{state}[i] = \texttt{true} \;\leftrightarrow\; q_i \in Q$

```
void epsClose(FSM* fsm, bool states[]) {
    bool change = true;
    while (change) {
       change = false;
       for (unsigned i = 0; i < fsm->numberStates; ++i)
       {
           if (states[i] && fsm->symbol[i] == 0) {
               if (!states[fsm->next1[i]]) {
                   change = true;
                   states[fsm->next1[i]] = true;
               }
               if (!states[fsm->next2[i]]) {
                   change = true;
                   states[fsm->next2[i]] = true;
               }
           }
       }
    }
}
```

Simulation der Potenz–Mengen–Konstruktion

```
bool simulate(FSM* fsm, char* word) {
    bool currentStates[fsm->numberStates];
    bool nextStates[fsm->numberStates];
    currentStates[0] = true;
    for (unsigned i = 1; i < fsm->numberStates; ++i) {
        currentStates[i] = false;
    }
    epsClose(fsm, currentStates);
    while (*word != 0) {
        for (unsigned i = 0; i < fsm->numberStates; ++i)
            nextStates[i] = false;
        for (unsigned i = 0; i < fsm->numberStates; ++i) {
            if (    currentStates[i]
                &&  fsm->symbol[i] == *word)
            {
                nextStates[fsm->next1[i]] = true;
            }
        }
        for (unsigned i = 0; i < fsm->numberStates; ++i)
            currentStates[i] = nextStates[i];
        epsClose(fsm, currentStates);
        ++word;
    }
    return currentStates[fsm->numberStates - 1];
}
```

Zahl der Zustände: n, Länge des Wortes: m

1. Komplexität `epsClose`: $\mathcal{O}(n^2)$
2. Komplexität `simulate`: $\mathcal{O}(m * n^2)$

Konstruktion RegExp $\mapsto$ FSM: Implementierung

1. Speicherplatz reservieren

```
FSM* allocateFSM(unsigned n) {
    FSM* fsm = malloc( sizeof(FSM) );
    fsm->numberStates = n;
    fsm->symbol       = malloc( n * sizeof(int) );
    fsm->next1        = malloc( n * sizeof(int) );
    fsm->next2        = malloc( n * sizeof(int) );
    return fsm;
}
```

2. Fsm f2 an Stelle o in f1 kopieren

```
void move(FSM* f1, FSM* f2, unsigned o) {
    for (unsigned i = 0; i < f2->numberStates; ++i)
    {   f1->symbol[i + o] = f2->symbol[i];
        f1-> next1[i + o] = f2-> next1[i] + o;
        f1-> next2[i + o] = f2-> next2[i] + o;
    }
}
```

3. $i \xrightarrow{\varepsilon} j$, $i \xrightarrow{\varepsilon} k$

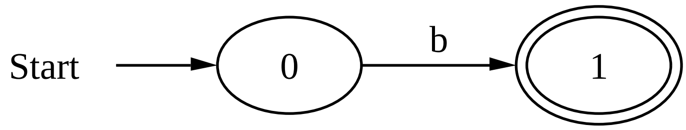
```
void epsTransition(FSM* fsm, unsigned i,
                   unsigned j, unsigned k) {
    fsm->symbol[i] = 0;
    fsm-> next1[i] = j;  fsm-> next2[i] = k;
}
```

4. $i \xrightarrow{c} j$

```
void charTransition(FSM* fsm, unsigned i,
                    unsigned j, char c) {
    fsm->symbol[i] = c;
    fsm-> next1[i] = j;  fsm-> next2[i] = j;
}
```

FSM zur Erkennung von Buchstaben

Akzeptieren des Buchstaben `b`



1. 2 Zustände

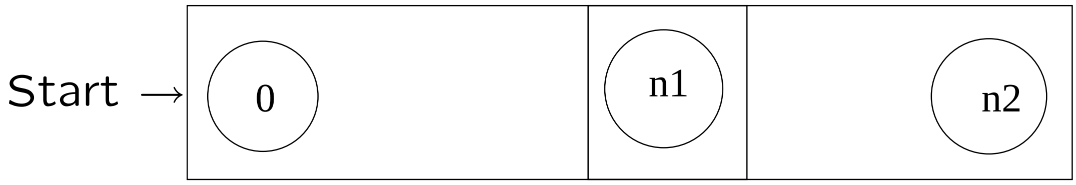
2. $0 \xrightarrow{b} 1$

3. $1 \xrightarrow{\varepsilon} 1$

Implementierung:

```
FSM* createCharacter(char c) {
    FSM* fsm = allocateFSM(2);
    charTransition(fsm, 0, 1, c);
    epsTransition (fsm, 1, 1, 1);
    return fsm;
}
```

Implementierung der Konkatenation

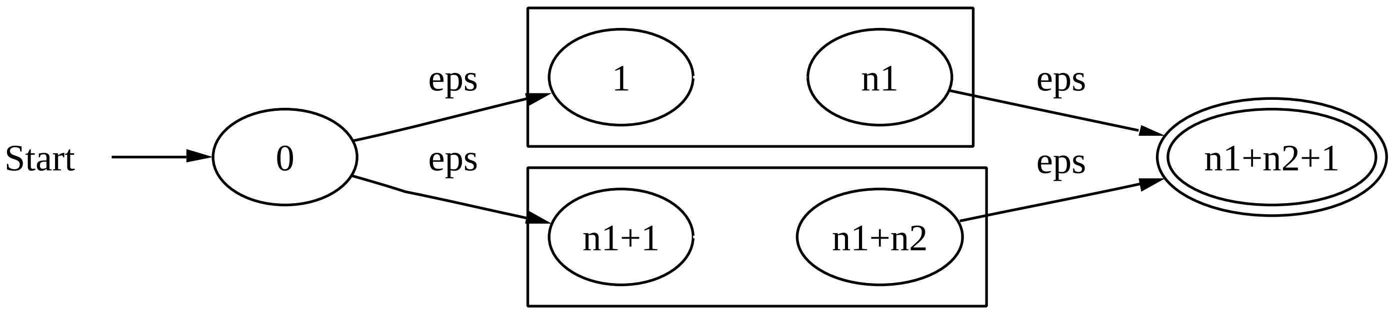


Implementierung

1. Größe Fsm `f1`: `n1`.
2. Größe Fsm `f2`: `n2`.
3. Größe neue Fsm: `n1 + n2 - 1`
4. Kopiere `f1` an Offsett 0 in neuer Fsm.
5. Kopiere `f2` an Offsett `n1-1` in neuer Fsm.

```
FSM* concat(FSM* f1, FSM* f2) {
    unsigned n1 = f1 ->numberStates;
    unsigned n2 = f2 ->numberStates;
    unsigned n  = n1 + n2 - 1;
    FSM* fsm = allocateFSM(n);
    move(fsm, f1, 0);
    move(fsm, f2, n1 - 1);
    freeFsm(f1);
    freeFsm(f2);
    return fsm;
}
```

Implementierung der Alternative

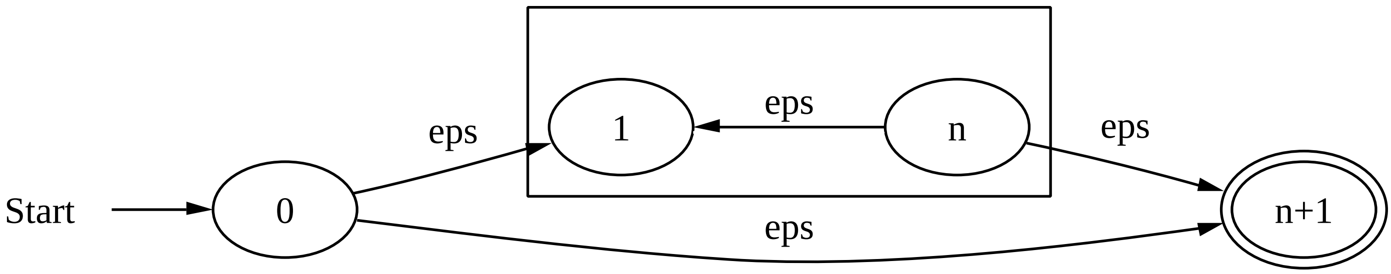


Implementierung

1. Größe Fsm `f1`: `n1`.
2. Größe Fsm `f2`: `n2`.
3. Größe neue Fsm: `n1 + n2 + 2`
4. Kopiere `f1` an Offsett 1 in neuer Fsm.
5. Kopiere `f2` an Offsett `n1+1` in neuer Fsm.

```
FSM* alternative(FSM* f1, FSM* f2) {
    unsigned n1 = f1 ->numberStates;
    unsigned n2 = f2 ->numberStates;
    unsigned n  = n1 + n2 + 2;
    FSM* fsm = allocateFSM(n);
    epsTransition(fsm, 0, 1, n1 + 1);
    move(fsm, f1, 1);
    move(fsm, f2, n1 + 1);
    epsTransition(fsm, n1, n - 1, n - 1);
    epsTransition(fsm, n - 2, n - 1, n - 1);
    epsTransition(fsm, n - 1, n - 1, n - 1);
    freeFsm(f1);   freeFsm(f2);
    return fsm;
}
```

Implementierung des Abschlusses



1. Größe Fsm `f`: `n`.
2. Größe neue Fsm: `n + 2`
3. Kopiere `f` an Offsett 1 in neuer Fsm.

Implementierung

```
FSM* closure(FSM* f) {
    unsigned n = f->numberStates;
    FSM* fsm = allocateFSM(n+2);
    epsTransition(fsm, 0, 1, n + 1);
    move(fsm, f, 1);
    epsTransition(fsm, n, n + 1, 1);
    epsTransition(fsm, n + 1, n + 1, n + 1);
    freeFsm(f);
    return fsm;
}
```

Konstruktion einer `RegExp` aus einer FSM

Definitionen: Sei Σ ein endliches Alphabet.

1. Es seien $\mathcal{L}_1, \mathcal{L}_2 \subseteq \Sigma^*$. Wir definieren die *Konkatenation* von $\mathcal{L}_1$ und $\mathcal{L}_2$ als:
$$\mathcal{L}_1\mathcal{L}_2 := \bigl\{vw \mid v \in \mathcal{L}_1 \wedge w \in \mathcal{L}_2\bigr\}$$

2. Sei $\mathcal{L} \subseteq \Sigma^*$. Für $n \in \mathbb{N}$ definieren wir die *Potenz* $\mathcal{L}^n$ durch Induktion über n:

   I.A. $n \mapsto 0$: $\qquad \mathcal{L}^0 := \{\varepsilon\}$

   I.S. $n \mapsto n+1$: $\qquad \mathcal{L}^{n+1} := \mathcal{L}^n\mathcal{L}$

3. Sei $\mathcal{L} \subseteq \Sigma^*$. Wir definieren den *Abschluß* von $\mathcal{L}$ als
$$\mathcal{L}^* = \bigcup_{n=0}^{\infty} \mathcal{L}^n$$

4. Seien $u, w \in \Sigma^*$. u ist *Präfix* von w, g.d.w. es gibt $v \in \Sigma^*$ mit $v \neq \varepsilon$ und $uv = w$. Schreibweise: $u \prec w$
$$u \prec w \;\Leftrightarrow\; \exists v \in \Sigma^* : v \neq \varepsilon \wedge uv = w.$$

Beispiele: Sei $\Sigma = \{a, b\}$, $\mathcal{L}_1 = \{ba, b\}$ und $\mathcal{L}_2 = \{abb, bb\}$.

1. $\mathcal{L}_1\mathcal{L}_2 = \{baabb, babb, babb, bbb\}$

2. $\mathcal{L}_1^3 = \{bbb, bbba, bbab, babb, babab, babba, bbaba, bababa\}$

3. $abbabb \prec abbabbabb$

Weitere Definitionen

Definition: Gegeben sei eine FSM

$$F = \langle \Sigma, \{q_1, \cdots, q_n\}, A, q_1, \texttt{next} \rangle$$

Wir definieren eine partielle Funktion

$$\texttt{next}^{(k)} : Z \times \Sigma^* \to Z$$

1. Fall: $\forall v \in \Sigma^* \backslash \{\varepsilon\} : v \prec w \to \texttt{next}(q, v) \in \{q_1, \cdots, q_k\}$

   Dann setzen wir

   $$\texttt{next}^{(k)}(q, w) = \texttt{next}^*(q, w).$$

2. Fall: $\exists v \in \Sigma^* \backslash \{\varepsilon\} : v \prec w \wedge \texttt{next}(q, v) \notin \{q_1, \cdots, q_k\}$

   Dann sei $\texttt{next}^{(k)}(q, w)$ undefiniert, wir setzen

   $$\texttt{next}^{(k)}(q, w) = \uparrow.$$

Falls $\texttt{next}^{(k)}(q, w) = p$ ist, so sagen wir, dass die Berechnung von w im Zustand q, die Zustände aus $\{q_{k+1}, \cdots, q_n\}$ *vermeidet*.

Für $k \in \{0, 1, \cdots, n\}$ und $i, j \in \{1, \cdots, n\}$ definieren wir Sprachen $\mathcal{R}_{i,j}^{(k)}$

$$\mathcal{R}_{i,j}^{(k)} := \left\{ w \in \Sigma^* \mid \texttt{next}^{(k)}(q_i, w) = q_j \right\}$$

Bemerkung: Es gilt

$$\mathcal{R}_{i,j}^{(n)} := \left\{ w \in \Sigma^* \mid \texttt{next}^*(q_i, w) = q_j \right\}$$

Konstruktion einer `RegExp` aus einer FSM

Gegeben: FSM $F = \langle \Sigma, \{q_1, \cdots, q_n\}, A, q_1, \texttt{next} \rangle$

Gesucht: $r \in \texttt{RegExp}$ mit $\mathcal{L}(r) = \mathcal{L}(F)$

Wir geben induktive Konstruktionen der Sprachen $\mathcal{R}_{i,j}^{(k)}$:

1. Induktions–Anfang: $k = 0$

   (a) $\exists a \in \Sigma : \texttt{next}(q_i, a) = q_j$ und $i \neq j$

   $$\mathcal{R}_{i,j}^{(0)} := \{a\}$$

   (b) $\exists a \in \Sigma : \texttt{next}(q_i, a) = q_i$

   $$\mathcal{R}_{i,i}^{(0)} := \{a, \varepsilon\}$$

   (c) $\forall a \in \Sigma : \texttt{next}(q_i, a) \neq q_j$ und $i \neq j$

   $$\mathcal{R}_{i,j}^{(0)} := \{\}$$

   (d) $\forall a \in \Sigma : \texttt{next}(q_i, a) \neq q_i$ und $i = j$

   $$\mathcal{R}_{i,i}^{(0)} := \{\varepsilon\}$$

2. Induktions–Schritt: $k \mapsto k + 1$

   $$\mathcal{R}_{i,j}^{(k+1)} := \mathcal{R}_{i,j}^{(k)} \cup \mathcal{R}_{i,k+1}^{(k)} \left(\mathcal{R}_{k+1,k+1}^{(k)}\right)^* \mathcal{R}_{k+1,j}^{(k)}$$

   Begründung: Es ist $w \in \mathcal{R}_{i,j}^{(k+1)}$ g.d.w.

   (a) $w \in \mathcal{R}_{i,j}^{(k)}$ oder

   (b) $w = x v_1 v_2 \cdots v_n y$ mit $n \geq 0$ und

   - $x \in \mathcal{R}_{i,k+1}^{(k)}$, $\quad y \in \mathcal{R}_{k+1,j}^{(k)}$
   - $v_i \in \mathcal{R}_{k+1,k+1}^{(k)}$ für alle $i = 1, \cdots, n$

Abschluß der Konstruktion

Wir definieren reguläre Ausdrücke $r_{i,j}^{(k)}$ mit

$$\mathcal{L}(r_{i,j}^{(k)}) = \mathcal{R}_{i,j}^{(k)}$$

1. Induktions–Anfang: $k = 0$
   (a) $\exists a \in \Sigma : \texttt{next}(q_i, a) = q_j$ und $i \neq j$
   $$r_{i,j}^{(0)} := a$$
   (b) $\exists a \in \Sigma : \texttt{next}(q_i, a) = q_i$
   $$r_{i,i}^{(0)} := a + \varepsilon$$
   (c) $\forall a \in \Sigma : \texttt{next}(q_i, a) \neq q_j$ und $i \neq j$
   $$r_{i,j}^{(0)} := \emptyset$$
   (d) $\forall a \in \Sigma : \texttt{next}(q_i, a) \neq q_i$
   $$r_{i,i}^{(0)} := \varepsilon$$
2. Induktions–Schritt: $k \mapsto k + 1$
   $$r_{i,j}^{(k+1)} := r_{i,j}^{(k)} + r_{i,k+1}^{(k)} \left(r_{k+1,k+1}^{(k)} \right)^* r_{k+1,j}^{(k)}$$

Setze $r_{i,j} := r_{i,j}^{(n)}$.

Sei $A = \{q_l, \cdots, q_n\}$ Menge der akzeptierenden Zustände.

$$r_F := r_{1,l} + r_{1,l+1} + \cdots + r_{1,n}.$$

Dann gilt

$$\mathcal{L}(F) = \mathcal{L}\big(r_F\big)$$

Ein Beispiel

Definition: Eine deterministische FSM

$$F = \langle \Sigma, Z, A, s_0, \texttt{next} \rangle$$

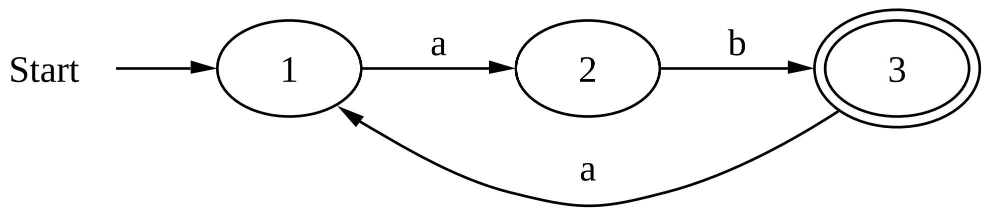
ist eine *partielle* FSM, wenn die Funktion `next` teilweise undefiniert ist.

Bemerkung: Die Konstruktion eines regulären Ausdrucks r mit

$$\mathcal{L}(r) = \mathcal{L}(F)$$

funktioniert auch für partielle FSMs.

Aufgabe: Berechnen Sie für folgende FSM einen äquivalenten regulären Ausdruck.



Nützliche Vereinfachungs–Regeln

1. $(\varepsilon + r)^* \simeq r*$
2. $(\varepsilon + r)(\varepsilon + r)^* \simeq r^*$
3. $r_1 + r_1 r_2^* \simeq r_1 r_2^*$
4. $r_1 + r_1 r_2 r_2^* \simeq r_1 r_2^*$
5. $r_1 (r_2 r_1)^* r_2 = r_1 r_2 (r_1 r_2)^*$

Berechnung des regulären Ausdrucks

	$k=0$	$k=1$	$k=2$	$k=3$
$r_{1,1}^{(k)}$	ε	ε	ε	$(aba)^*$
$r_{1,2}^{(k)}$	a	a	a	$a(baa)^*$
$r_{1,3}^{(k)}$	$\emptyset$	$\emptyset$	ab	$ab(aab)^*$
$r_{2,1}^{(k)}$	$\emptyset$	$\emptyset$	$\emptyset$	$ba(aba)^*$
$r_{2,2}^{(k)}$	ε	ε	ε	$(baa)^*$
$r_{2,3}^{(k)}$	b	b	b	$b(baa)^*$
$r_{3,1}^{(k)}$	a	a	a	$a(aba)^*$
$r_{3,2}^{(k)}$	$\emptyset$	aa	aa	$aa(baa)^*$
$r_{3,3}^{(k)}$	ε	ε	$\varepsilon + aab$	$(aab)^*$

1. Spalte: folgt unmittelbar aus Diagramm
2. Spalte: einzige Änderung dort, wo Pfade der Länge 2 sind, deren mittlerer Knoten 1 ist:

   $3 \xrightarrow{a} 1 \xrightarrow{a} 2$, also $r_{3,2}^{(1)} = aa$
3. Spalte: einzige Änderung dort, wo Pfade mit Knoten 2 hinzukommen:

   $3 \xrightarrow{a} 1 \xrightarrow{a} 2 \xrightarrow{b} 3$, also $r_{3,3}^{(2)} = \varepsilon + aab$

   $1 \xrightarrow{a} 2 \xrightarrow{b} 3$, also $r_{1,3}^{(2)} = ab$
4. Spalte: Sei $i \xrightarrow{u} j$ und $j \xrightarrow{v} j$

   $r_{i,j}^{(3)} = uv^*$

Also: $\mathcal{L}(F) = \mathcal{L}\big(ab(aab)^*\big)$

Pumping Lemma

Definition: Für ein Wort $w \in \Sigma^*$ bezeichnet

$$|w|$$

die *Länge* (Zahl der Buchstaben) von w.

Für $n \in \mathbb{N}$ definieren wir w^n induktiv:

I.A. $n \mapsto 0$: $\quad w^0 := \varepsilon$

I.S. $n \mapsto n+1$: $\quad w^{n+1} := ww^n$

Definition: Sei Σ Alphabet. Eine Sprache $\mathcal{L} \subseteq \Sigma^*$ ist *regulär der Ordnung* n g.d.w. es einen endl. Automaten

$$F = \langle \Sigma, Z, A, s_0, \mathtt{next} \rangle$$

gibt, so daß gilt: $\quad \mathcal{L} = \mathcal{L}(F)$ und $\mathtt{card}(Z) = n$.

Hier bezeichnet $\mathtt{card}(Z)$ die Zahl der Zustände.

Satz (*Pumping Lemma*)

Vor.: $\mathcal{L}$ ist reguläre Sprache der Ordnung n.

Beh.: Für alle $w \in \mathcal{L}$ mit $|w| \geq n$ gibt es $x, y, z \in \Sigma^*$ mit

1. $w = xyz$
2. $|xy| \leq n$
3. $|y| \geq 1$
4. $\forall k \in \mathbb{N} : xy^kz \in \Sigma^*$

Beweis des Pumping Lemma

Sei $\mathcal{L} = \mathcal{L}(F)$ mit $\quad F = \langle \Sigma, Z, q_0, \texttt{next} \rangle$ und $n = \texttt{card}(Z)$.

Sei $w = a_1 a_2 \cdots a_m \in \mathcal{L}(F)$ mit $m \geq n$. Dann gibt es Berechnung der FSM F mit Label w:

$$q_0 \stackrel{a_1}{\rightarrow} q_1 \stackrel{a_2}{\rightarrow} q_2 \stackrel{a_3}{\rightarrow} \cdots \stackrel{a_m}{\rightarrow} q_m \qquad \text{und } q_m \in A$$

Es können nicht alle Zustände in der Menge

$$\{q_0, q_1, q_2, \cdots, q_n\}$$

verschieden sein, es gibt also $i, j \in \{0, \cdots, n\}$ mit

$$i < j \qquad \text{und} \qquad q_i = q_j.$$

Wir definieren

$$x := a_1 \cdots a_i, \qquad y := a_{i+1} \cdots a_j, \qquad z := a_{j+1} \cdots a_m$$

Dann haben wir die folgenden Berechnungen der FSM F

1. $q_0 \stackrel{a_1}{\rightarrow} \cdots \stackrel{a_i}{\rightarrow} q_i$, also $q_0 \stackrel{x}{\rightarrow} q_i$
2. $q_i \stackrel{a_{i+1}}{\rightarrow} \cdots \stackrel{a_j}{\rightarrow} q_i$, also $q_i \stackrel{y}{\rightarrow} q_i$
3. $q_i \stackrel{a_{j+1}}{\rightarrow} \cdots \stackrel{a_m}{\rightarrow} q_m$, also $q_i \stackrel{z}{\rightarrow} q_m$

Damit gilt auch

$$q_0 \stackrel{x}{\rightarrow} \underbrace{q_i \stackrel{y}{\rightarrow} q_i \stackrel{y}{\rightarrow} \cdots \stackrel{y}{\rightarrow} q_i}_{k} \stackrel{z}{\rightarrow} q_m$$

und das zeigt, dass

$$xy^k z \in \mathcal{L} \qquad \text{für alle } k \in \mathbb{N}$$

Aus $i < j$ und $|y| = j - i$ folgt $|y| \geq 1$.

Wegen $|xy| = j$ und $j \in \{0, \cdots, n\}$ folgt auch

$$|xy| \leq n. \qquad \square$$

Anwendung des Pumping Lemma

Aufgabe: Sei $\Sigma = \{\mathtt{a}, \mathtt{b}\}$. Zeigen Sie, dass die Sprache

$$\mathcal{L} := \big\{w \in \Sigma^* \mid \mathtt{nr}(w, \mathtt{a}) = \mathtt{nr}(w, \mathtt{b})\big\}$$

nicht regulär ist.

Ein Wort in $\mathcal{L}$ enthält genauso viele Buchstaben $\mathtt{a}$ wie $\mathtt{b}$.

Beweis indirekt.

Annahme: Sei $\mathcal{L}$ reguläre Sprache der Ordnung n.
Betrachte das Wort $w := a^n b^n$. Es gilt

$$a^n b^n \in \mathcal{L} \quad \text{und} \quad |a^n b^n| = 2 * n.$$

Also gibt es Worte x, y und z mit:

1. $w = a^n b^n = xyz$
2. $|xy| \leq n$
3. $|y| \geq 1$
4. $xy^k z \in \mathcal{L}$

Aus $\quad |xy| \leq n \quad$ und $\quad xyz = a^n b^n \quad$ folgt

$$xy \prec a^n$$

und daraus folgt

$$\mathtt{nr}(x, \mathtt{b}) = 0 \quad \text{und} \quad \mathtt{nr}(y, \mathtt{b}) = 0.$$

Fortsetzung der Aufgabe

Aus $xyz \in \mathcal{L}$ folgt wegen

$$\mathtt{nr}(xyz, \mathtt{a}) = \mathtt{nr}(x, \mathtt{a}) + \mathtt{nr}(y, \mathtt{a}) + \mathtt{nr}(z, \mathtt{a}) \text{ und}$$

$$\mathtt{nr}(xyz, \mathtt{b}) = \mathtt{nr}(x, \mathtt{b}) + \mathtt{nr}(y, \mathtt{b}) + \mathtt{nr}(z, \mathtt{b}) = \mathtt{nr}(z, b)$$

die Gleichung

$$\mathtt{nr}(x, \mathtt{a}) + \mathtt{nr}(y, \mathtt{a}) + \mathtt{nr}(z, \mathtt{a}) = \mathtt{nr}(z, b) \qquad \text{(A)}$$

Ebenso folgt aus $xy^2z \in \mathcal{L}$ und

$$\mathtt{nr}(xy^2z, \mathtt{a}) = \mathtt{nr}(x, \mathtt{a}) + 2 * \mathtt{nr}(y, \mathtt{a}) + \mathtt{nr}(z, \mathtt{a}) \text{ und}$$

$$\mathtt{nr}(xy^2z, \mathtt{b}) = \mathtt{nr}(x, \mathtt{b}) + 2 * \mathtt{nr}(y, \mathtt{b}) + \mathtt{nr}(z, \mathtt{b}) = \mathtt{nr}(z, b)$$

die Gleichung

$$\mathtt{nr}(x, \mathtt{a}) + 2 * \mathtt{nr}(y, \mathtt{a}) + \mathtt{nr}(z, \mathtt{a}) = \mathtt{nr}(z, b) \qquad \text{(B)}$$

Ziehen wir (A) von (B) ab, so erhalten wir

$$\mathtt{nr}(y, a) = 0$$

Weil auch

$$\mathtt{nr}(y, b) = 0$$

gilt, folgt

$$y = \varepsilon$$

Das steht im Widerspruch zu

$$|y| \geq 1 \ . \qquad \square$$

Erkenntnis: Endliche Automaten können nicht zählen!

Reguläre Ausdrücke in der Praxis: `sed`

Zeichen mit Sonderbedeutung bei `sed`:

1. “.” : paßt auf jeden Buchstaben außer Zeilenumbruch.
   “`a.c`” paßt auf “`abc`”, “`azc`”, “`a1c`”, ···

2. “*” : Quantor
   beliebig viele Wiederholungen.
   “`fo\*`” paßt auf “`f`”, “`fo`”, “`foo`”, “`fooo`”, etc.

3. “\+” : Quantor
   mindestens eine Wiederholung.
   “`fo\+`” paßt auf “`fo`”, “`foo`”, “`fooo`”, ···

4. “\?” : Quantor
   keine oder eine Wiederholung.
   “`fo\?`” paßt auf “`f`” und “`fo`”.

5. “\|: Auswahl von zwei Möglichkeiten
   “`eins\|zwei`” paßt sowohl auf “`eins`” als auch auf “`zwei`”

6. “^” paßt auf leeren String am Zeilen–Anfang
   “`^int`” paßt auf “`int`” am Zeilen–Anfang.

7. “$” paßt auf leeren String am Zeilen–Ende.
   “`^$`” paßt auf Leerzeile.

Reguläre Ausdrücke

8. “[” und “]” begrenzen *Zeichen–Mengen*

   “`[ad]`” paßt auf “`a`” und “`d`”.

   Spezifikation von *Intervallen* durch “-”

   (a) “`[a-z]`” paßt auf jeden Klein–Buchstaben.

   (b) “`[a-z*.]`” paßt auf jeden Klein–Buchstaben und auf “*” und “.”

   Merke: “*”, “+”, “?”, “.” verlieren Sonderbedeutung in Zeichen–Mengen.

   (c) “^”: Komplement einer Zeichen–Mengen.

   “`[^0-9]`” paßt auf alle Zeichen, die keine Ziffern sind.

9. “\(” und “\)”: Gruppierung

   “`\(ja\)\+`” paßt auf

   “`ja`”, “`jaja`”, “`jajaja`”, $\cdots$

   “`\(J\|j\)a`” paßt auf

   “`Ja`” und “`ja`”

Anwendung: `sed`

`sed s/`*regexp*`/`*replacement*`/g`

replacement kann “\1”, “\2”, “\3” $\cdots$ “\3” enthalten.

“\\n” steht für n-te Klammer

```
sed 's/\\emph{\([^}]\*\)}/<em>\1<\/em>/g'
```

ersetzt “`\emph`{*string*}” durch “*string*”.

Ein komplexeres Beipspiel `sed`

Gegeben: Datei der Form

```
8 5
9 3
10 3
11 9
12 3
...
```

Idee: Datei spezifiziert Primzahlen

$n \quad m$ spezifiziert $2^n - m$

Automatisches Ausrechnen mit `sed` und `bc`

```
cat prim.txt | \
    sed 's/\([0-9]\+\) \([0-9]\+\)/2^\1 - \2/g' | \
    bc
```

Andere Kommandos, die mit regulären Ausdrücken arbeiten

1. `grep` *regexp* *file*$_1$ $\cdots$ *file*$_n$
2. *XEmacs*:
   (a) `isearch-forward-regexp`
   (b) `query-replace-regexp`
3. `find` [*directory*] `-regexp` *regexp-pattern* [*action*]
4. Grundlage von *Perl*, *Tcl*, *Python*