Formale Grundlagen der Informatik

1

Motivation
Formale Sprachen
Zustände in der Informatik
Endliche Automaten

Einführung und Motivation

Theoretische Informatik - Motivation

- *Wiki:* Gebiet der Informatik, das sich mit der **Abstraktion**, **Modellierung** und grundlegenden **Fragestellungen** beschäftigt, die die Struktur, Verarbeitung, Übertragung und Wiedergabe von Information betreffen.

➢ Mathematische Methoden
  - Darstellung der Abstraktion
  - Nachweis von Eigenschaften
  - exakte Beantwortung der Fragestellungen

Oft sind Dinge nicht so, wie sie auf den ersten Blick erscheinen!

- *Abstraktion hilft!*
- In diesem Kurs : „Problemgetriebenes“ Vorgehen

Verwendung abstrakter Modelle

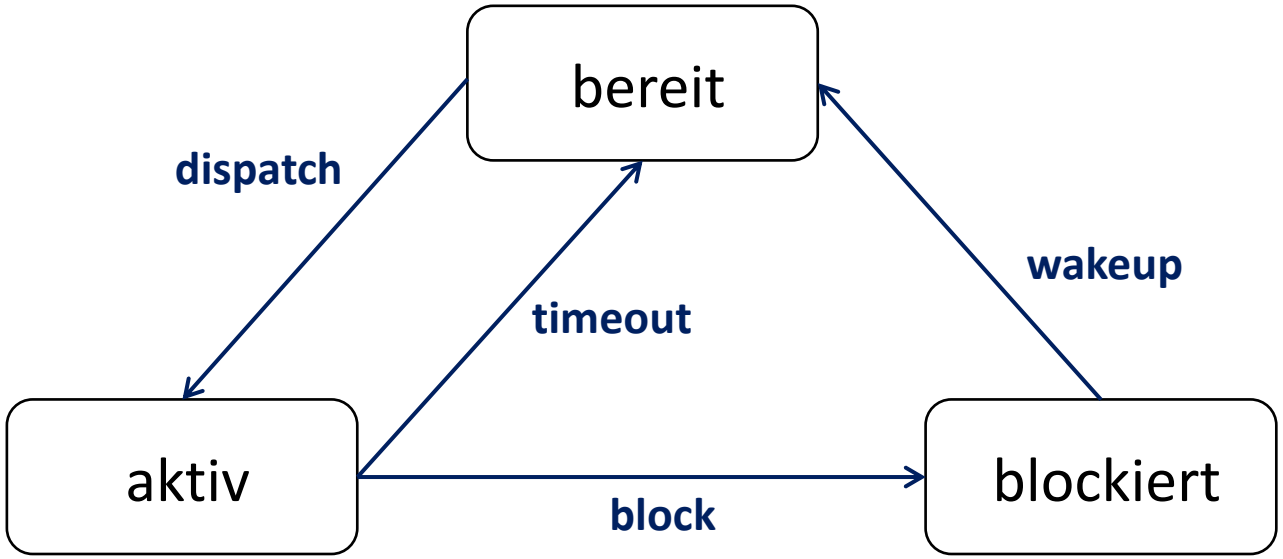
- **Algorithmen** basieren auf solchen Modellen
  - Beweise für Korrektheit und Terminieren
  - ermöglichen Implementierung
- **Compiler/Interpreter** sind im Wesentlichen Implementierungen von Modellen der Theoretischen Informatik
- Grenzen der Berechenbarkeit
  - generell
  - aufgrund der Komplexität
- **Software Engineering**
  (Qualität im Entwicklungsprozess und im Softwareprodukt)
- …

Softwarequalität

- **Entwurf** → Verwendung von **Modellen**: *Abstraktion hilft!*
  - bei der Kommunikation im Entwicklerteam und mit Stakeholdern
  - um Wiederverwendbarkeit festzustellen (*s. Entwurfsmuster*)
  - z.B. (UML-)Struktur- und Verhaltensdiagramme
- **Software-Test / Formale Verifikation**
  - *Warum wird so selten formal verifiziert?* → **Grenzen der Berechenbarkeit**
  - Formale Beweise im Einzelfall → Automatisierung / **Theorembeweiser**
- **Statische Analysen** … basierend auf geeigneten Modellen
  - Möglichkeiten und Grenzen verschiedener Modellierungssprachen
  - Analysemethoden und deren Korrektheit, …

Systemzustände: Prozesse

- Softwareprozesse führen zu einer Folge von Systemzuständen
- *Beispiel*: Prozesszustände (vereinfacht)



Welche Befehlsfolgen sind valide?

Welche Befehlsfolgen führen zu welchen Systemzuständen?

Beispiel: Python-Programme

Programm:

```
x = 7                                  a
y = input("Eingabe: ")                 b
y = int(y)                             c
if (y > 0):
      while(y >= x):
            y = y - x                  d
else:
      y = 0                            e
```

Programmläufe:

für y = 1: **abc**

für y = 8: **abcd**

für y = 15: **abcdd**

System-/Programmzustände:
Werte der Programmvariablen

Beispiel: Python-Programme

Programm:

```
x = 7                                  a
y = input("Eingabe: ")                 b
y = int(y)                             c
if (y > 0):
      while(y >= x):
            y = y - x                  d
else:
      y = 0                            e
```

Programmläufe:

für y = 1: **abc**

für y = 8: **abcd**

für y = 15: **abcdd**

Beschreibung aller Programmläufe?

$$\{ abcd^n \mid n \geq 0 \} \cup \{abce\}$$

Algorithmen und ihre Darstellung

Algorithmen: Folgen von Anweisungen

- plattformunabhängiger Entwurf oder Analyse oft in informaler Sprache (Text) oder in semi-formaler Sprache (Diagramme, Pseudocode, …)
- Implementierung in **formaler Sprache** (Programmiersprache)

Formale Sprache

- feste, präzise Syntax
- eindeutige Semantik
- kann von Maschinen interpretiert und ausgeführt werden

Formale Sprachen

Alphabete und Wörter

- Endliche Menge von Anweisungen → endliche Menge von Symbolen
- Ein **Alphabet** ist eine endliche Menge.
  Bezeichnung durch Großbuchstaben, oft Σ
- Die Elemente eines Alphabets heißen **Symbole** (oder **Buchstaben)**.
- Ein **Wort** ist eine endliche Folge von Buchstaben.
  Bezeichnung durch Kleinbuchstaben vom Ende des Alphabets: $u, v, w, x, y, z, w_1, w_2, \ldots$

- Die **Länge** eines Wortes w ist die Anzahl seiner Buchstaben: $|w|$
  Beispiel: $|abcdd| = 5$
- Das **leere Wort** der Länge 0 wird mit ε bezeichnet: $|\varepsilon| = 0$

Vorkommen von Buchstaben in Wörtern

- Sei Σ ein Alphabet, $w \in \Sigma$ und $U \subseteq \Sigma$.
- Dann bezeichnet $|w|_U$ die Anzahl der Vorkommen von Buchstaben aus U im Wort w.
- *Beispiel:* $\Sigma = \{a, b, c\}$, $U = \{a, c\}$
  - $|abacbbc| = 7$, $|abacbbc|_U = 4$,
  - $|abacbbc|_a = 2$, $|abacbbc|_b = 3$.

Konkatenation von Wörtern

- Seien u und v zwei Wörter über einem Alphabet Σ.
  Ihre **Konkatenation $u \cdot v$** ist das Wort uv.
- *Beispiel:* Für $u = ab$ und $v = ba$ ist $u \cdot v = abba$
- Die Konkatenation ist *nicht kommutativ*: $v \cdot u = baab$
- Das leere Wort ist *neutrales Element* bei der Konkatenation:
  Für jedes Wort w gilt: $w \cdot \varepsilon = \varepsilon \cdot w = w$
- Σ^* sei die Menge aller Wörter über Σ,
  Σ^+ sei die Menge aller nichtleeren Wörter über Σ.

➢ $\Sigma^* = \Sigma^+ \cup \{\varepsilon\}$

Potenzen von Wörtern

Rekursive Definition: Sei w ein Wort. Dann ist

- $w^0 = \varepsilon$ und
- $w^i = w^{i-1} \cdot w,$ für alle $i \geq 1$.

  - $w^1 = w^0 \cdot w = \varepsilon \cdot w = w$

Σ^* und Σ^+ als Trägermengen

- Σ^* und Σ^+ sind **abgeschlossen** unter Konkatenation

➢ $(\Sigma^*, \cdot)$ und $(\Sigma^+, \cdot)$ sind algebraische Strukturen

- Konkatenation ist **assoziativ**: Für alle $u, v, w \in \Sigma^*$ gilt $(uv)w = u(vw)$

➢ $(\Sigma^*, \cdot)$ und $(\Sigma^+, \cdot)$ sind **Halbgruppen**

➢ $(\Sigma^*, \cdot, \varepsilon)$ ist ein **Monoid** (Halbgruppe mit neutralem Element)

Formale Sprachen

- Sei Σ ein Alphabet.
- Eine **formale Sprache L** über Σ ist eine Menge von Wörtern über Σ:

$$L \subseteq \Sigma^*$$

- Die Mengen $\emptyset$ und $\{\varepsilon\}$ sind Sprachen. Es gilt $\emptyset \neq \{\varepsilon\}$.
- Eine **endliche Sprache** ist eine Menge mit endlich vielen Wörtern:

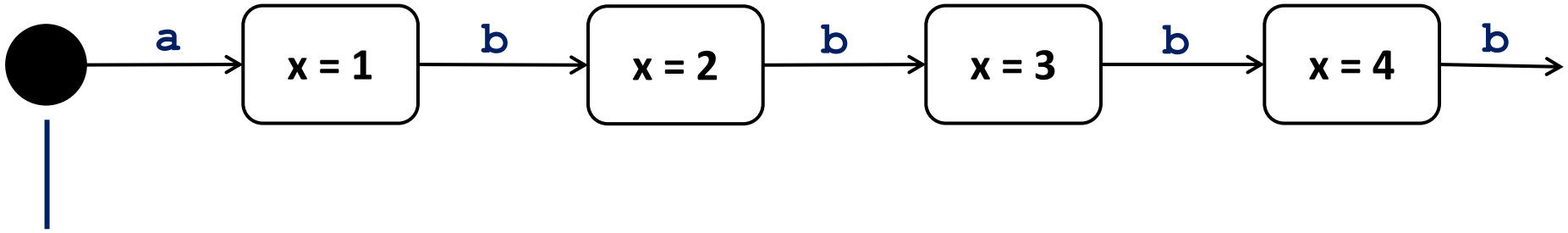
$$L = \{abc, abcd, abcdd\}$$

- Relevante Sprachen sind oft *unendlich*.
  Wir suchen endliche Beschreibungen für diese unendlichen Objekte.
  - ➢ verschiedene Modelle der Theoretischen Informatik
  - ➢ Beispiel: Endliche Automaten

Endliche Automaten

Zustandsräume

```
x = 1                a
while (x > 0):
      x = x + 1      b
```



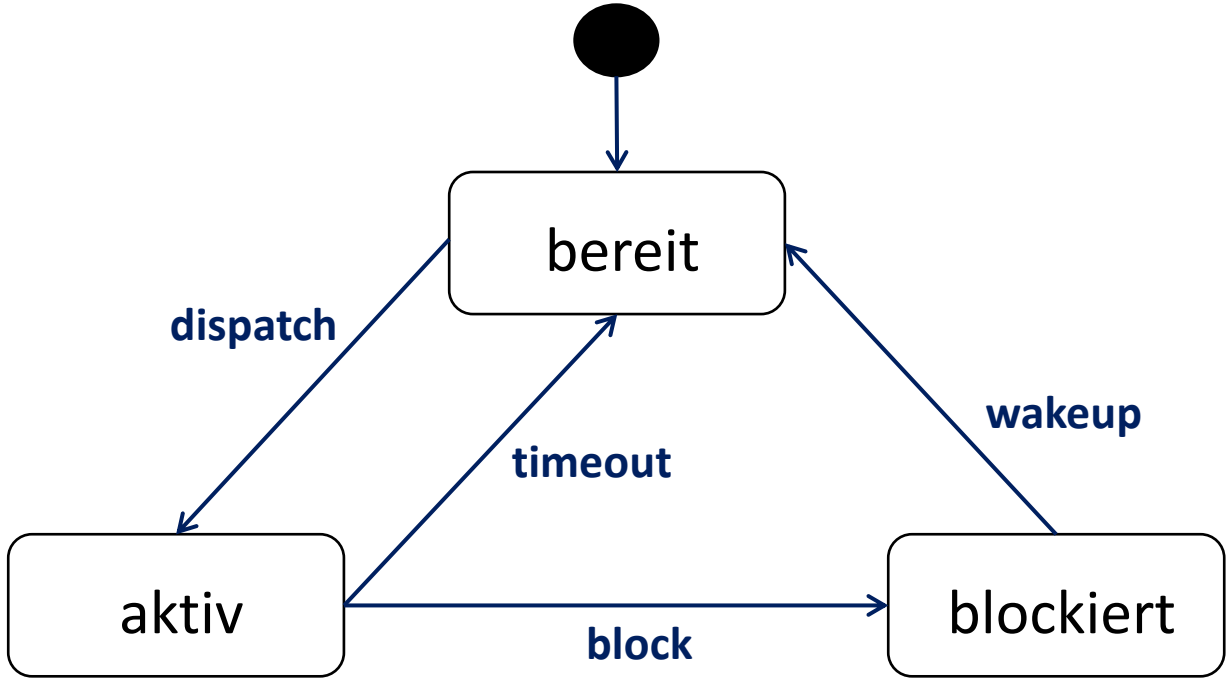
Startzustand q_0 :
alle Variablen undefiniert

Zustandsraum: **Menge aller Zustände**:

$$\{q_0\} \cup \{\, \text{'x} = n\text{'} \mid n \geq 1 \,\}$$

Endliche Zustandsräume

- Prozesszustände als UML-Zustandsdiagramm:

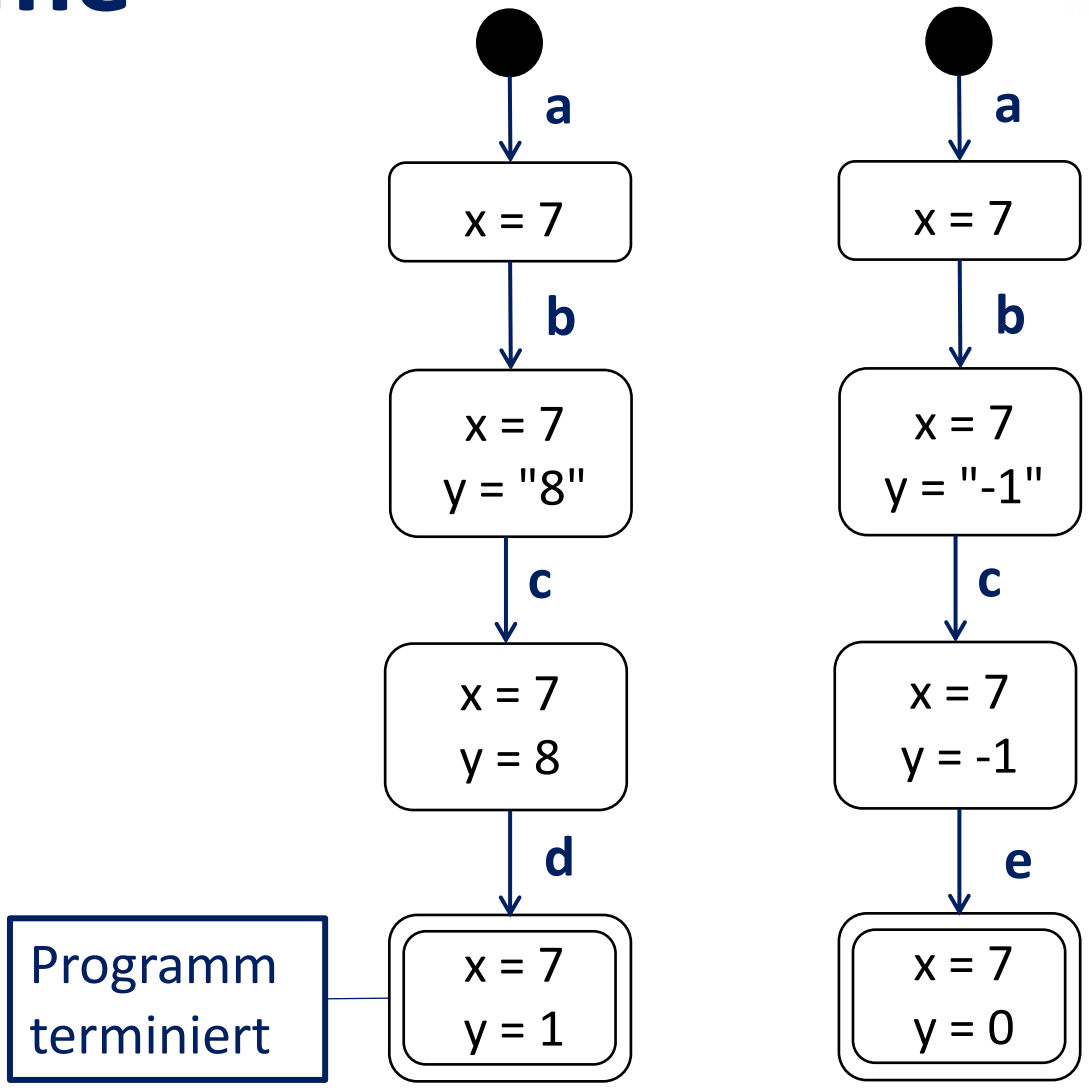


- *weitere Beispiele:*
  - Geld-/Verkaufsautomaten
  - Verkehrsampel
  - elektronische Schaltungen, z.B. Flip-Flop (*2 Zustände*)
  - Client-Server-Systeme
  - GUIs
  - Geschäftsprozesse
  - …

Endliche Zustandsräume

Programm:

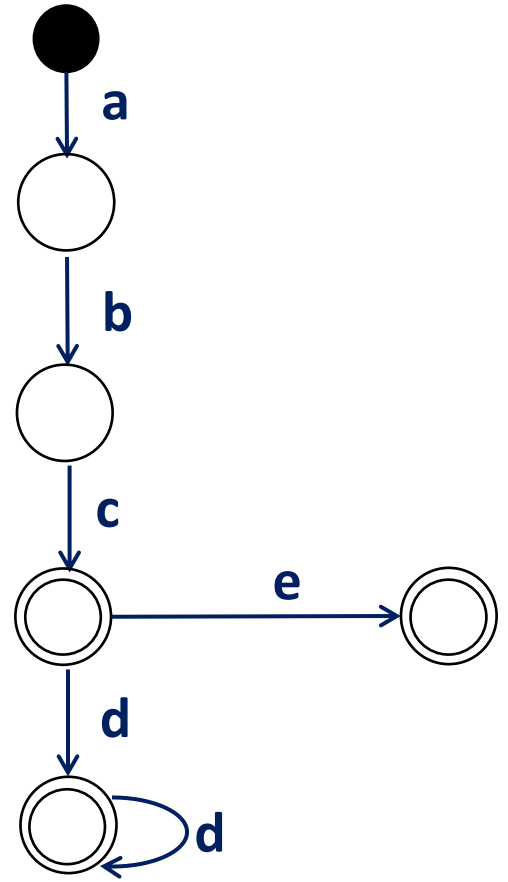
```
x = 7                              a
y = input("Eingabe: ")             b
y = int(y)                         c
if (y > 0):
      while(y >= x):
            y = y - x              d
else:
      y = 0                        e
```



Alle Programmläufe als Zustandsdiagramm

Programm:

```
x = 7                              a
y = input("Eingabe: ")             b
y = int(y)                         c
if (y > 0):
      while(y >= x):
            y = y - x              d
else:
      y = 0                        e
```



$\{ abcd^n \mid n \geq 0 \} \cup \{abce\}$

Abstraktion zu einheitlichem Modell

benötigen:

- **Zustandsmenge:** endliche Menge von Zuständen
- **Eingabealphabet:** endliche Menge von Eingabezeichen, die die Zustandsübergänge auslösen (Aktionen/Befehle)
- **Überführungsfunktion:** ordnet jedem Zustand bei jedem Eingabezeichen den Nachfolgezustand zu
- **Startzustand:** Zustand bei Start des Systems
- **Menge der Zustände**, die Eingabefolgen als korrekt **akzeptieren**

Deterministischer Endlicher Automat

Definition: Ein **deterministischer endlicher Automat (DEA)** ist ein 5-Tupel $A = (Q, \Sigma, \delta, q_0, F)$, wobei

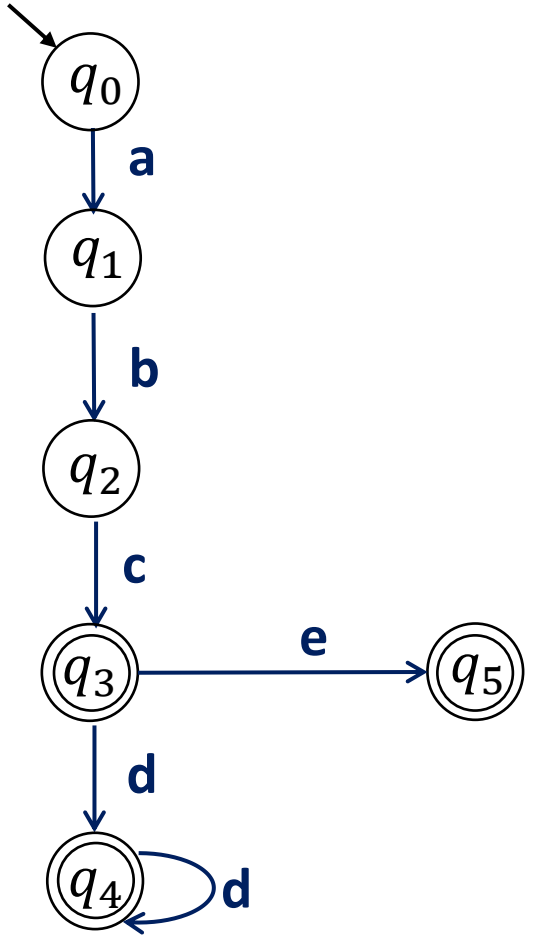
- Q eine endliche Menge ist, deren Elemente Zustände heißen,
- Σ ein Alphabet ist, dessen Elemente Eingabesymbole heißen,
- $\delta: (\mathrm{Q} \times \Sigma) \rightarrow Q$ die Überführungsfunktion ist,
- $q_0 \in Q$ der Startzustand ist und
- $F \subseteq Q$ die Menge der akzeptierenden Zustände ist.

DEA - unser Beispiel

$A = (Q, \Sigma, \delta, q_0, F)$ mit

- $Q = \{q_0, q_1, q_2, q_3, q_4, q_5\}$
  repräsentiert durch Knoten
- $\Sigma = \{a, b, c, d, e\}$
  repräsentiert durch Kantenmarkierungen
- $\delta(q_0, a) = q_1$ $\quad \delta(q_1, b) = q_2$ $\quad \delta(q_2, c) = q_3$
  $\delta(q_3, d) = q_4$ $\quad \delta(q_3, e) = q_5$ $\quad \delta(q_4, d) = q_4$
  repräsentiert durch Kanten
- $q_0 \in Q$ (ein mit einem Pfeil markierter Knoten)
- $F = \{q_3, q_4, q_5\}$ (durch „Doppelkreis“ markierte Knoten)

?



Überführungsfunktion

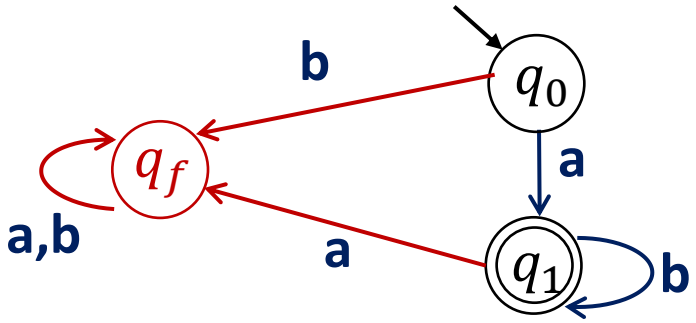
- Formal ist $\delta: (\mathrm{Q} \times \Sigma) \rightarrow Q$ als totale Funktion definiert.
- Kann erreicht werden durch Einführung eines **Fehlerzustands q_f** :
  - $Q = \{q_0, q_1, q_2, q_3, q_4, q_5, q_f\}$
  - $\delta(q_0, a) = q_1$ $\quad \delta(q_1, b) = q_2$ $\quad \delta(q_2, c) = q_3$
    $\delta(q_3, d) = q_4$ $\quad \delta(q_3, e) = q_5$

    $\delta(q, x) = q_f$ für alle anderen Paare $(q, x) \in Q \times \Sigma$

    $\rightarrow \delta(q_f, x) = q_f$ für alle $x \in \Sigma$

DEA – Beispiel 1

Der Automat soll alle (Programmläufe)/Eingabefolgen/Eingabewörter beschreiben/akzeptieren, bei denen auf ein a beliebig viele b folgen.



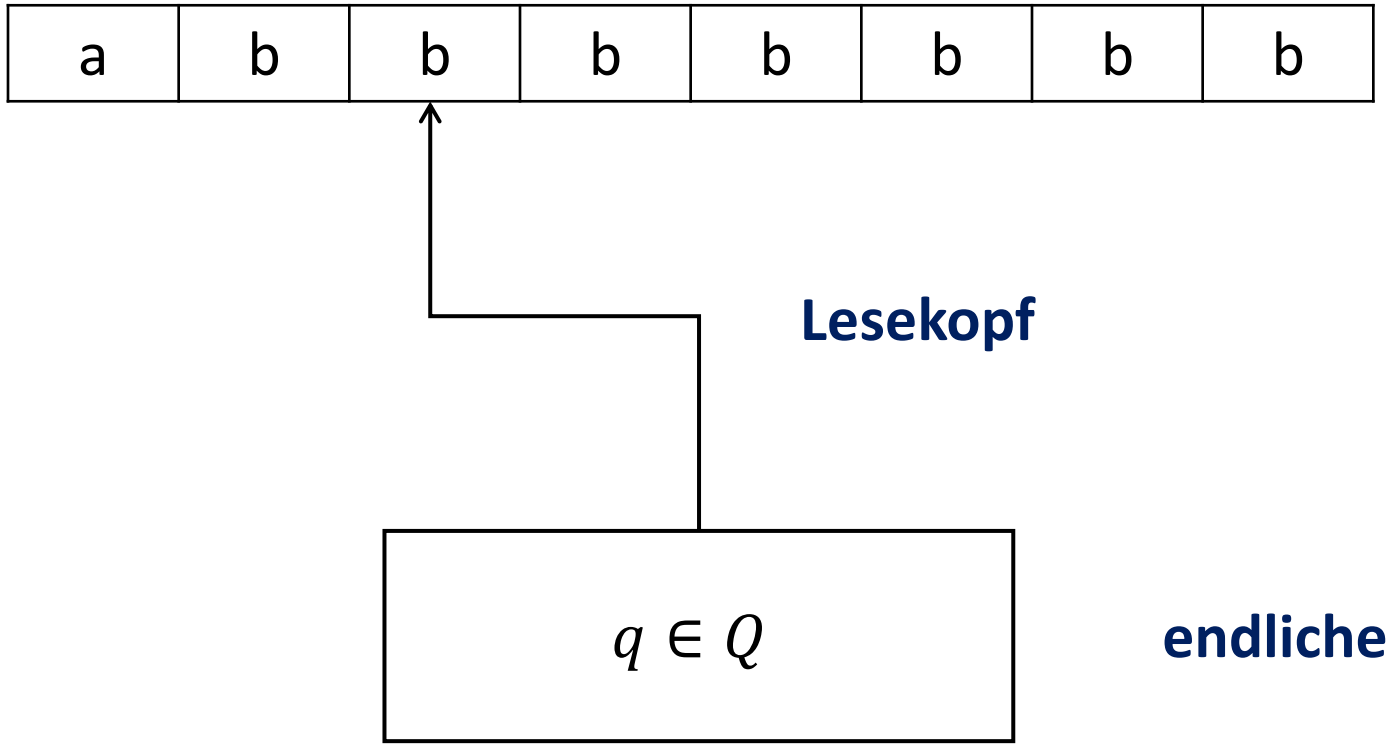
Transitionsgraph von A_1

$$A_1 = (\{q_0, q_1, q_f\}, \{a, b\}, \delta_1, q_0, \{q_1\})$$

Überführungstabelle für δ_1:

	a	b
→ q_0	q_1	q_f
∗ q_1	q_f	q_1
q_f	q_f	q_f

DEA als Maschine

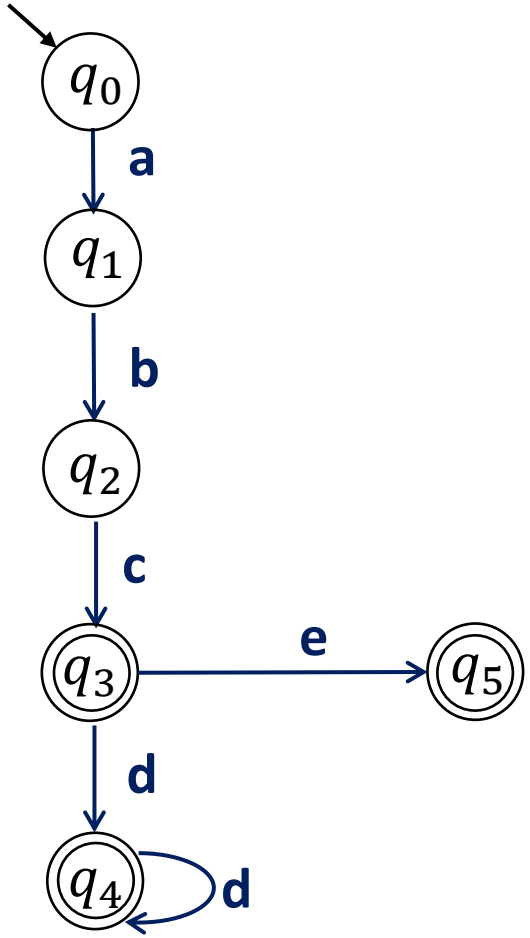


beginnt über dem ersten Buchstaben der Eingabe im Startzustand

DEA – akzeptierte Sprache

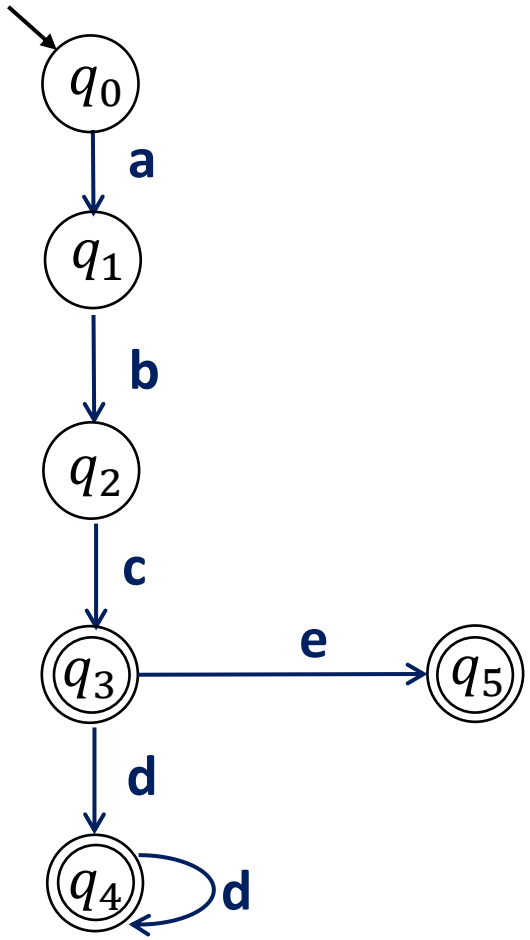
- *intuitiv:* Menge aller Eingabefolgen (Wörter über Σ), die den Automaten vom Startzustand in einen akzeptierenden Zustand überführen
- *am Beispiel:* **abc, abd, abdd, abddd, …, abce**
- $L(A) = \{ abcd^n \mid n \geq 0 \} \cup \{abce\}$
- *formale Definition?!*
  - *benötigen Erweiterung von δ auf Wörter:*

    $\delta: (Q \times \Sigma) \rightarrow Q \quad \longrightarrow \quad \hat{\delta}: (Q \times \Sigma^*) \rightarrow Q$
  - $\hat{\delta}(q_0, a) = q_1 \qquad \hat{\delta}(q_0, ab) = q_2 \qquad \hat{\delta}(q_0, abc) = q_3$



DEA – erweiterte Überführungsfunktion

- $\hat{\delta}(q_0, a) = q_1 \qquad \hat{\delta}(q_0, ab) = q_2 \qquad \hat{\delta}(q_0, abc) = q_3$
- $\hat{\delta}(q_0, abc) = \delta(q_2, c)$, da $\hat{\delta}(q_0, ab) = q_2$
- $\hat{\delta}(q_0, ab) = \delta(q_1, b)$, da $\hat{\delta}(q_0, a) = q_1$
- $\hat{\delta}(q_0, a) = \delta(q_0, a)$, da $\hat{\delta}(q_0, \varepsilon) = q_0$



DEA – erweiterte Überführungsfunktion

Definition: Sei $A = (Q, \Sigma, \delta, q_0, F)$ ein DEA.

Die **erweiterte Überführungsfunktion** $\hat{\delta}: (\mathrm{Q} \times \Sigma^*) \to Q$ ist rekursiv wie folgt definiert:

Für alle $q \in Q$, $a \in \Sigma$ und $w \in \Sigma^*$ sei

- $\hat{\delta}(q, \varepsilon) \quad = q$
- $\hat{\delta}(q, wa) = \delta(p, a)$ mit $p = \hat{\delta}(q, w)$
  $= \delta(\hat{\delta}(q, w), a)$

DEA – akzeptierte Sprache

- *intuitiv:* Menge aller Eingabefolgen (Wörter über Σ), die den Automaten vom Startzustand in einen akzeptierenden Zustand überführen
- *am Beispiel:* **abc, abd, abdd, abddd, …, abce**
- $L(A) = \{\, abcd^n \mid n \geq 0 \,\} \cup \{abce\}$
- **Definition:** Sei $A = (Q, \Sigma, \delta, q_0, F)$ ein DEA.
  Die von A **akzeptierte Sprache $L(A)$** ist die Sprache
  $L(A) = \{\, w \mid \hat{\delta}(q_0, w) \in F \,\}$.