HOCHSCHULE REGENSBURG
UNIVERSITY OF APPLIED SCIENCES

Informatik (Bachelor) 2. Semester

Grundlagen der Informatik*

Hefter des Sommersemesters 2009

Alexander Thaller

stand 21. April 2009

Aus der Vorlesung von Professor Dr. rer. nat. Jorg Striegnitz (Str)

*Gefundene Fehler oder Verbesserungsvorschlge bitte hier http://fhrein2ss09.codeplex.com/WorkItem/List.aspx berichten oder alternativ mir eine E-Mail schreiben alexander.thaller@stud.fh-regensburg.de. Vielen Dank.

Inhaltsverzeichnis

Übersicht

Professor: Jorg Striegnitz

Homepage: http://homepages.fh-regensburg.de/ stj39817/index.html

E-Mail: joerg.striegnitz@informatik.fh-regensburg.de

Zeitplan:

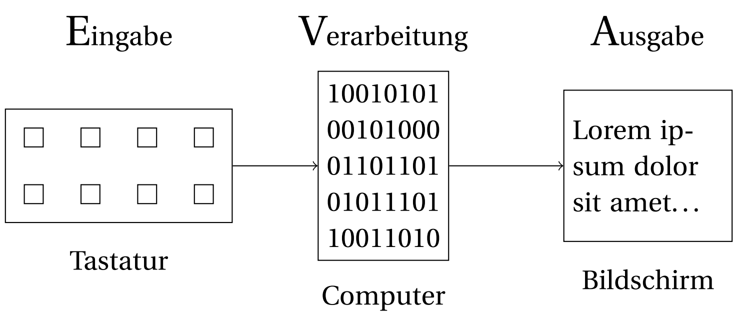
- 4 Stunden pro Woche
- 2 Stunden Vorlesung
- 2 Stunden Übungen

Klausur: Zulassungsvorrausetzungen: 50% in den Übungen und eine Aufgabe Vorrechnen

Tutoren:

- Herr Bauer (Mo. 4. Stunde)
- Herr Lainer (Do. 6. Stunde)
- Frau Sporrer (U4M)
- Frau Franziska Brandstetter (Di. 2. Stunde) (franziska.brandstetter@web.de)
- Herr Brillner (U612) (billner.michael@web.de)

1 Grundlagen



Ziel: Formale Darstellung der Ein-/Ausgaben eines Computers

$\Rightarrow$ Repräsentation von Daten

Anmerkung: Computerprogramme sind ebenfalls Daten.

1.1 Alphabet

Def.! (Def.!) 1.1. Eine endliche Menge, die nicht leer ist, von Symbolen Σ heißt Alphabet. Die Elemente von Σ heißen Buchstaben (Zeichen, Symbole). □

Beispiel:

- $\Sigma_{\text{RGB}} = \left\{\underline{\text{rot}}, \underline{\text{grün}}, \underline{\text{blau}}\right\}$
- $\Sigma_{\text{Bool}} = \{0, 1\}$ das Boole'sche Alphabet
- $\Sigma_{\text{lat}} = \{a, b, c, \ldots, z\}$ das lateinische Alphabet
- $\Sigma_{\text{Tastatur}} = \left\{A, B, \ldots, Z, \textvisiblespace, <, >, (,), \ldots\right\}$
- $\Sigma_{\text{m}} = \{0, \ldots, m-1\}$ für $m \geq 1$ die m-adische Darstellung einer Zahl.
- $\Sigma_{\text{logic}} = \{0, 1, x, (,), \vee, \wedge, \neg\}$ ein Alphabet für logische Ausdrücke

1.2 Wort

Def.! 1.2 (Wort)**.** Sei Σ ein Alphabet. Ein Wort über Σ ist eine endliche (eventuell leere) Folge von Buchstaben. Das leere Wort ϵ (oder manchmal auch λ) ist die leere Buchstabenfolge. Die Menge aller Wörter über Σ bezeichnen wir mit Σ^*. Ferner sei $\Sigma^+ = \Sigma^* - \{\epsilon\}$. Fortan gehen wir davon aus, dass $\epsilon \notin \Sigma$ gilt. □

Beispiel: $\overbrace{(0,1,0,1,0,1,0)}^{Z}$ ist ein Wort über Σ_{Bool}, Σ_{Tastatur} und Σ_{logic}.

- Z ist <u>Wort</u> über Σ_{Bool}
- Z ist aus Σ^*_{Bool}

ϵ ist ein Wort über einem **bel.!** (**bel.!**) Alphabet ($\epsilon \in \Sigma^*$ für alle Alphabete Σ^*)

Notation 1.1. Wörter werden künftig ohne Kommata geschrieben und lassen auch die Klammern weg.
Also **zB!** (**zB!**) 01010 statt (0,1,0,1,0)
Beachte: Unterschied Wort TI (Folge von Symbolen) und Wort „Deutsch“

1.2.1 Wortlänge

Def.! 1.3 (Wortlänge)**.** Sei Σ ein Alphabet und $w \in \Sigma^*$. Die Wortlänge $|w|$ eines Wortes w ist die Länge des Wortes als Folge. Für $a \in \Sigma$ bezeichnet $|w|_a$ die Anzahl der Vorkommen von a in w. □

Beispiel:

- $|001001| = 6$
- $|001001|_1 = 2$
- $|\epsilon| = 0$
- $|\textvisiblespace| = 1$

1.3 Konkatenation und Verkettung

Def.! 1.4 (Konkatenation/Verkettung)**.** Die Konkatenation für ein Alphabet Σ ist eine Abbildung (oder auch Funktion) K:

$$\Sigma^* \times \Sigma^* \to \Sigma^*, \text{ so dass} K(u, v) = uv$$

für alle $u, v \in \Sigma^*$. Anstelle von $K(u, v)$ schreiben wir $u \cdot v$. □

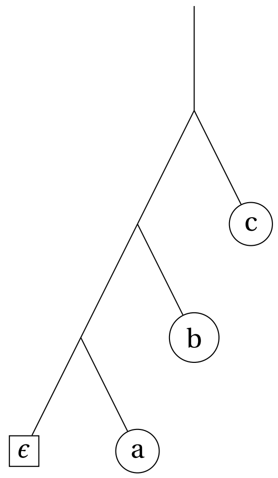
Beobachtung: Konkatenation ist assoziativ. $\Rightarrow (a \cdot b) \cdot c = a(b \cdot c)$
Ferner gilt: $\epsilon \cdot w = w = w \cdot \epsilon$ Monoid.

1.4 Induktive Definitionen

1.4.1 Induktive Definition des Wortbegriffes

Die Menge aller Wörter über Σ (Σ^*) ist induktiv definiert durch:

1. $\epsilon \in \Sigma^*$
2. sei $w \in \Sigma^*$ und $a \in \Sigma$, so $w \cdot a \in \Sigma^*$



1.4.2 Induktive Definition der Wortlänge

1. für $w = \epsilon$ sei $|w| = 0$

2. für $w = v \cdot x$ ist $|w| = |v| + 1$

$$\begin{aligned}
\Big|\underbrace{\epsilon ab}_{v}\underbrace{c}_{x}\Big| &= \Big|\underbrace{\epsilon a}_{v}\underbrace{b}_{x}\Big| + 1 \\
&= \Big|\underbrace{\epsilon}_{v}\underbrace{a}_{x}\Big| + 1 + 1 \\
&= |\epsilon| + 1 + 1 + 1 \\
&= 0 + 1 + 1 + 1 \\
&= 3
\end{aligned}$$

1.5 Kanonische Ordnung

Um alle Wörter aus Σ^* systematisch aufzuzählen, ordnen wir Alphabet und Wörter.

Def.! 1.5 (Kanonische Ordnung)**.** Sei $\Sigma = \{a_1, a_2, \ldots, a_n\}$ und $<: \Sigma \times \Sigma$ eine Ordnung auf Σ mit $a_1 < a_2 < a_3 < \cdots < a_n$. Wir definieren die kanonische Ordnung auf Σ^* wie folgt:

$$u < v \textbf{ gdw.! } (\textbf{gdw.!})\ |u| < |v| \vee$$

$$|u| = |v| \wedge u = w a_i u_1 \wedge v w a_j u_2 \text{ für } u, v \in \Sigma^*\ w, u_1, u_2 \in \Sigma^* \text{ und } a_i < a_j (i < j)$$

□

1.6 Teilwort, Präfixe, Suffixe

Def.! 1.6 (Teilwort, Präfixe, Suffixe)**.** Sei Σ ein Alphabet und seien $v, w \in \Sigma^*$

- v heißt Teilwort von w **gdw.!** $\exists s, t \in \Sigma^*: w = svt$
- v heißt Suffix von w **gdw.!** $\exists s \in \Sigma^*: w = sv$

  w

s	v

- v heißt Präfix von w **gdw.!** $\exists t \in \Sigma^*: w = vt$

v	t

- $v \neq \epsilon$ heißt echtes Teilwort (Präfix/Suffix) von w **gdw.!** $v \neq w$ und v ist Teilwort (Präfix/-Suffix) von w.

□

1.7 Wortumkehr

Def.! 1.7 (Wortumkehr)**.** Sei Σ ein Alphabet und sei $w = a_1 \cdot a_2 \cdot a_3 \ldots a_n$ mit $a_i \in \Sigma (1 \leq i \leq n)$ ein Wort. Dann ist die Wortumkehr $w^{\mathscr{R}}$ von w **def.!** (**def.!**) durch $w^{\mathscr{R}} = a_n a_{n-1} \ldots a_2 a_1$ □

1.7.1 Alternativ: induktive Definition

- $\epsilon^{\mathscr{R}} = \epsilon$
- falls $v = w \cdot a$, so $v^{\mathscr{R}} = a w^{\mathscr{R}}$

Beispiel: $(\overline{a}\overline{b}\overline{c})^{\mathscr{R}}$ $v = \overline{a}\overline{b}\overline{c}$ $w = \overline{a}\overline{b}$ $a = \overline{c}$

$$
\begin{aligned}
\Rightarrow v^{\mathscr{R}} &= \overline{c} \cdot \underbrace{(\overline{a}\overline{b})^{\mathscr{R}}} \; v = \overline{a}\overline{b} w = \overline{a} a = \overline{b} \\
\Rightarrow v^{\mathscr{R}} &= \overline{c}\overline{b} \cdot (\overline{a})^{\mathscr{R}} \; v = \epsilon \cdot \overline{a} w = \epsilon a = \overline{a} \\
v^{\mathscr{R}} &= \overline{c} \cdot \overline{b} \cdot \overline{a} \cdot (\epsilon)^{\mathscr{R}} \\
&= \overline{c} \cdot \overline{b} \cdot \overline{a} \cdot \epsilon \\
&= \overline{c} \cdot \overline{b} \cdot \overline{a}
\end{aligned}
$$

1.8 Iteration

Def.! 1.8 (Iteration)**.** Sei Σ ein Alphabet. Für alle Wörter $w \in \Sigma^*$ und $i \in \mathbb{N}$ definieren wir die i-te Iteration w^i als

$$w^0 = \epsilon, w^1 = w$$

und

$$w^i = w \cdot w^{i-1}$$

Beispiel:

$$
\begin{aligned}
aabb \cdot bbab &= aabbbbab \\
&= a^2 b^4 ab \\
ababc &= (ab)^2 c
\end{aligned}
$$

1.9 Sprachen

Def.! 1.9 (Sprache, Konkatenation von Sprachen, Kleene-Stern)**.** Sei Σ ein Alphabet. Eine Sprache L ist eine Teilmenge von Σ^* $(L \subseteq \Sigma^*)$ (potentiell unendlich!). Das Komplement $L^{\mathscr{C}}$ der Sprache L bezüglich Σ^* ist die Sprache $\Sigma * -L$. $L_\emptyset$ ist die leere Sprache; $L_\epsilon = \{\epsilon\}$. Seien L_1 und L_2 Sprachen, dann ist

$$L_1 \cdot L_2 = L_1 L_2 = \left\{u \cdot v \in (\Sigma_1 \cup \Sigma_2)^x | u \in L_1 \wedge v \in L_2\right\}$$

die Konkatenation von L_1 und L_2. Ferner definieren wir für eine Sprache L

$$L^0 = L_\epsilon L^{i+1} = L \cdot L^i$$

und

$$L^* = \bigcup_{i \in \mathbb{N}} L^i \text{ und } L^+ = \bigcup_{i \in \mathbb{N}-\{0\}} L^i = L \cdot L^*$$

Beispiel:

- $L_1 = \emptyset$
- $L_2 = \{\epsilon\}$
- $L_3 = \{\epsilon, aa, bb, ab, ba\}$
- $L_4 = \{a, b\}^* = \{\epsilon, a, b, aa, ab, ba, bb, \ldots\}$
- $L_5 = \{a\}^* = \{\epsilon, a, aa, aaa, \ldots\} = \left\{a^0, a^1, a^2, \ldots\right\}$
- $L_6 = \left\{a^p | p = \text{ Primzahl}\right\}$
- $L_7 = \left\{a^i b^{2i} a^i | i \in \mathbb{N}\right\}$
- L_8 = alle syntaktisch korrekten *JAVA*-Programme.
- L_9 = alle typ-korrekter *JAVA*-Programme.
- L_{10} = alle terminierenden *JAVA*-Programme.

1. Sprachen geeignet zur Darstellung aller Ein/Ausgaben?
2. Reicht Σ_{Bool}?
3. Effizienter Test, ob $w \in L$ gilt. $\boxed{\leftarrow\text{Automaten}}$
4. Wie kann man Sprachen formal definieren? $\boxed{\leftarrow\text{Grammatiken}}$
5. Sind Sprachen aufzählbar? $\boxed{\leftarrow\text{Berechenbarkeit}}$
   Effizient möglich. $\boxed{\leftarrow\text{Komplexitätstheorie}}$

Lemma 1.10. Seien L_1, L_2 und L_3 Sprachen über einem Alphabet Σ. Dann gilt

$$L_1 \cdot L_2 \cup L_1 \cdot L_3 = L_1 \cdot (L_2 \cup L_3)$$

Beweis 1.1.

- $L_1 \cdot L_2 \cup L_1 \cdot L_3 \subseteq L_1 \cdot (L_2 \cup L_3)$
  - $L_1 \cdot L_2 \subseteq L_1 \cdot (L_2 \cup L_3)$

$$\begin{aligned} L_1 \cdot L_2 &= \{uv | u \in L_1 \wedge v \in L_2\} \\ &\subseteq \{uv | u \in L_1 \wedge L_2 \cup L_3\} \\ &= L_1 \cdot (L_2 \cup L_3) \end{aligned}$$

  - $L_1 \cdot L_3 \subseteq L_1 \cdot (L_2 \cup L_3)$
    $\rightarrow$analog
- $L_1 \cdot (L_2 \cup L_3) \subseteq L_1 \cdot L_2 \cup L_1 \cdot L_3$

$$\begin{aligned} & x \in L_1 \cdot (L_2 \cup L_3) \text{ beliebig} \\ \Leftrightarrow & x \in \{uv | u \in L_1 \wedge v \in L_2 \cup L_3\} \text{ (Definition Konkatenation)} \\ \Leftrightarrow & \exists u \in L_1 \wedge \exists v \in L_2 \cup L_3 : x = uv \text{ (Übergang zur Prädikaten Logik)} \\ \Leftrightarrow & \exists u \in L_1 \wedge (\exists v \in L_2 \vee \exists v \in L_3) : x = uv \text{ (Übergang Prädikaten Logik 2)} \\ \Leftrightarrow & (\exists u \in L_1 \wedge \exists v \in L_2 : x = uv) \vee (\exists u \in L_1 \wedge \exists v \in L_3 : x = uv) \\ & \text{(Distributivgesetz der Prädikaten Logik)} \\ \Leftrightarrow & x \in \underbrace{\{uv | u \in L_1 \wedge v \in L_2\}}_{=L_1 \cdot L_2} \vee x \in \underbrace{\{uv | u \in L_1 \wedge v \in L_3\}}_{=L_1 \cdot L_3} \\ \Leftrightarrow & x \in L_1 \cdot L_2 \vee x \in L_1 \cdot L_3 \\ \Leftrightarrow & x \in L_1 \cdot L_2 \cup L_1 \cdot L_3 \end{aligned}$$

1.10 Sprachen zur Beschreibung von Problemen

1.10.1 Entscheidungsprobleme

Def.! 1.11. Das Entscheidungsproblem (Σ, L) für ein gegebenes Alphabet Σ und eine Sprache $L \in \Sigma^*$ ist, für jedes $w \in \Sigma^*$ zu entscheiden, ob

$$w \in L \vee w \notin L$$

$$\Sigma^*_{\text{Input}} \longrightarrow \boxed{\Sigma^*_{\text{Bool}} \rightarrow \Sigma^*_{\text{Bool}}} \longrightarrow \Sigma^*_{\text{Output}}$$

Ein Algorithmus löst das Entscheidungsproblem (Σ, L), falls $\forall w \in \Sigma^*$

$$A(w) = \begin{cases} 1 & w \in L \\ 0 & \text{sonst.} \end{cases}$$

Wir sagen auch, dass A L erkennt.

Def.! 1.12. Existiert für eine Sprache L ein Algorithmus, der diese erkennt, so heißt L rekursiv.

Beispiel: Ein wichtiges Entscheidungsproblem ist der Primzahltest.

$$\left(\Sigma_{\text{Bool}}, \{w \in \Sigma^*_{\text{Bool}} | \text{Zahlwert } (w) \text{ ist Primzahl}\}\right)$$

Beispiel: Das Erfüllbarkeitsproblem.

$$\left(\Sigma_{\text{logic}} m \left\{w \in \Sigma^*_{\text{logic}} | w \text{ erfüllbar}\right\}\right)$$

$$\neg (A_1 \vee A_2) \wedge A_3 \wedge A_4 \wedge \neg A_4$$

Ein ebenfalls wichtiges Entscheidungsproblem ist die Frage der Äquivalenz von Algorithmen A und B.

$$\left(\Sigma_{\text{Tastatur}}, \{B | B \text{ ist äquivalent zu } A\}\right)$$

A ist äquivalent zu B, falls A auf jede mögliche Eingabe genauso reagiert wie B und umgekehrt A.

$$\boxed{\text{A}} \xrightarrow{\text{Optimiert}} \boxed{\text{B}}$$

1.10.2 Aufzählungsalgorithmus für L

Def.! 1.13. Sei Σ ein Alphabet und $L \subseteq \Sigma^*$. A ist ein Aufzählungsalgorithmus für L, falls A für jede Eingabe $n \in \mathbb{N} - (0)$ die kanonisch (siehe Kanonische Ordnung) ersten n Wörter $w_0, w_1, \dots w_{n-1}$ der Sprache L ausgibt.

Lemma 1.14. Sei Σ ein Alphabet und $L \subseteq \Sigma^*$.

- L ist rekursiv $\overset{\textbf{gdw.!}}{\Leftrightarrow}$ es existiert Aufzählungsalgorithmus für L
- L entscheidbar

1.10.3 Funktionsberechnungen

Def.! 1.15 (Funktionsproblems)**.** Seien Σ_D und Σ_W zwei Alphabete. Wir sagen, dass ein Algorithmus A eine Funktion $f : \Sigma_D^* \to \Sigma_W^*$ berechnet (oder das Funktionsproblem f löst), falls

$$\forall w \in \Sigma_D^* : A(w) = f(w)$$

Existiert ein Algorithmus, der f berechnet, so heißt f berechenbar.

1.10.4 Relationsprobleme

Def.! 1.16. Seien Σ_1 und Σ_2 Alphabete und $R \subseteq \Sigma_1^* \times \Sigma_2^*$ eine Relation. Ein Algorithmus A berechnet R (oder löst das Relationsproblem R), falls für jedes $w \in \Sigma_1^*$ gilt:

$$(w, A(w)) \in R$$

1.11 Kolmogorov Komplexität

Beispiel (Wortlänge vs. Informationsgehalt)**:** Das Wort

$$w = 01010101010101$$

ist zwar länger als das Wort

$$v = 10101100$$

kann aber durch $(01)^7$ kompakter dargestellt werden. Es reichen offenbar fünf Zeichen um die Information in w zu kodieren.

Das Auffinden einer kürzeren Darstellung für ein Wort nennt man Komprimierung.

Ziel: Um den Informationsgehalt zweier Wörter zu vergleichen, könnten wir zunächst ein Standard-Komprimierungs Verfahren suchen.

Problem: Wie findet man ein Verfahren, welches immer die kleinstmögliche Darstellung garantiert?

Beispiel (Lauflängenkodierung)**:** Wir überführen eine Binärzahl zunächst in die Darstellung

$$b_1^{l_1} b_2^{l_2} \ldots, b_i \in \Sigma_{\text{Bool}}^*$$

wobei wir auch die Exponenten in Binärschreibweise darstellen. Von dieser Darstellung gehen wir wie folgt über zu einem Wort über $\Sigma_\sharp = \left(0, 1, \sharp\right)$

$$l_1 \sharp b_1 \sharp l_2 \sharp b_2 \ldots$$

Nach Σ_{Bool} kommen wir durch Anwendung eines Homomorphismus.
$\Sigma = \{0, 1, \sharp\}$

$$\left.\begin{array}{l} h(0) = 00 \\ h(1) = 11 \\ h(\sharp) = 10 \end{array}\right\} h : \Sigma_\sharp \to \Sigma^*_{\text{Bool}}$$

$$h(a_1 \dots a_n) = h(a_1) \cdot h(a_2) \cdot \dots \cdot h(a_n)$$

$$\begin{array}{rccccccc} & h(0) & \cdot & h(0) & \cdot & h(\sharp) & \cdot & h(1) \\ k(00\sharp 1) = & 00 & & 00 & & 01 & & 11 \\ & 0 & & 0 & & 10 & & 1 \end{array}$$

$H : \Sigma_1^* \to \Sigma_2^*$

- $h(\epsilon) = \epsilon$
- $h(u \cdot v) = h(u) \cdot h(v)$

Beispiel: Sei $w = (100)^{11}(101)^3(01)^3\, |w| = 43$
In unserer Zwischendarstellung ist dies

$$1011\sharp 100\sharp 11\sharp 101\sharp 11\sharp 01\sharp$$

und mit dem Homomorphismus $h(0) = 00$, $h(1) = 11$ und $h(\sharp) = 10$ gelangen wir zu

$$1100\,1111\,1011\,0000\,1011\,1110\dots \text{ mit } \left|w'\right| = 42$$

Für große Exponenten kann es sich lohnen, die Lauflängenkodierung auch auf diese Anzuwenden. Zum Beispiel:

$$(011)^{1048576} = (011)^{2^{20}}$$

Diese Strategie kann man beliebig fortsetzen, um zum Beispiel Wörter der Bauarten $(01)^{2^{2^n}}$ oder $(01)^{2^{2^{2^n}}}$ zu kodieren.
Allgemeines kompressions Verfahren wird durch anderen Umstand ebenfalls schwer auffindbar.

Beispiel (Codierung durch Primfaktorzerlegung)**:** Jede Zahl können wir als Folge von Primfaktoren eindeutig darstellen: zum Beispiel $884736 = 2^1 \cdot 3^3 \cdot 4^7$. Unter Einbeziehung der Basis können wir dies darstellen als.

$$\text{Bin}(e_1)\sharp\text{Bin}(p_1)\sharp\text{Bin}(e_2)\sharp\dots$$

Verfahren der Primfaktorzerlegung und der Lauflängenkodierung sind **unvergleichbar**. Eines der beiden Verfahren ist Grundsätzlich besser.

Def.! 1.17. Für alle Wörter $w \in \Sigma^*_{\text{Bool}}$ ist die Kolmogorov-Komplexität $K_c(w)$ definiert als die binäre Länge *des kürzesten `C++` Programmes*, welches w generiert.

Lemma 1.18. Es existiert eine Konstante d, so dass für jedes $w \in \Sigma^*_{\text{Bool}}$ gilt:

$$K_c(w) \leq |w| + d$$

Beweis 1.2. Für jedes w aus Σ^*_{Bool} nehmen wir folgendes C++ Programm:

$$P(\underbrace{w}_{\text{Zeichenkette}}) = \begin{cases} \texttt{int main(int args, char** args) \{} \\ \qquad \texttt{char * ws = w; //"abcdefg"} \\ \qquad \texttt{count++ w;} \\ \texttt{\}} \end{cases}$$

Regelmäßige Wörter haben eine geringere Kolmogorov-Komplexität. Betrachte **zB!** Wörter der Bauart $v_n = 0^n$ für $n \in \mathbb{N} \setminus \{0\}$

$$P(n) = \begin{cases} \texttt{unsigned int n;} \\ \texttt{for(int i = 1; i <= n; i++)} \\ \qquad \texttt{cout << "0";} \end{cases}$$

Bis auf die konstante n sind alle $P(n)$ identisch. Zur Kodierung von n als Binärzahl benötigt man $\lceil \text{ld}(n+1) \rceil$ Bits. Damit

$$K_c(v_n) \leq \lceil \text{ld}(n+1) \rceil + c \leq \lceil \text{ld}\, n \rceil + c' = \lceil \text{ld}\, |v_n| \rceil + c'$$

Betrachte $u_n = 0^{n^2}$ $n \in \mathbb{N} - \{0\}$

$$P(n) \begin{cases} \cancel{\texttt{m = n + n}} \quad \left| \begin{array}{l} \texttt{m = n;} \\ \texttt{m = m * m;} \\ \texttt{for(int i = 1; i <= m; i++)} \\ \qquad \texttt{cout << "0";} \end{array} \right. \end{cases}$$

$$K_c(u_n) \leq \lceil \text{ld}\, n \rceil + c \leq \lceil \text{ld} \sqrt{|u_n|} \rceil + c$$

$$|u_n| = n^2$$
$$\Leftrightarrow n = \sqrt{|u_n|}$$

Def.! 1.19. Sei $n \in \mathbb{N}$. Die Kolmogorov-Komplexität

$$K_c(n) = K_c(\text{Bin}(n)).$$

Lemma 1.20. Für jedes $n \in \mathbb{N}\backslash 0$ existiert ein Wort $w_n \in \Sigma^n_{Bool}$, so dass

$$K_c(w_n) \geq |w_n|$$

Es gibt Wörter der Länge n, die nicht komprimierbar sind.

Beweis 1.3. Wir haben genau 2^n Wörter $x_1, \ldots, x_{2^n} \in \Sigma^n_{\text{Bool}}$. Sei für $i = 1, \ldots, 2^n$ $\text{Prog}(x_i)$ der Maschinencode des `C++` Programmes von x_i und sei $K_c(x_i) = \left|\text{Prog}(x_i)\right|$ die Länge eines kürzesten Algorithmus. Es ist klar, dass für $x_i \neq x_j$ gilt:

$$\text{Prog}(x_i) \neq \text{Prog}(x_j)$$

wir haben also 2^n unterschiedliche Programme. Es genügt zu zeigen, dass eines dieser Programme die Länge n hat.

Jetzt hilft das kombinatorisches Argument: es ist unmöglich 2^n verschiedene Maschinencodes der Länge kleiner als n zu haben, denn die Anzahl aller unterschiedlichen, nicht leeren, Wörter aus $\Sigma^{n-1}_{\text{Bool}}$ ist

$$\sum_{i=1}^{n-1} 2^i = 2^n - 2 < 2^n$$

Also muss es unter den 2^n Programmen mindestens eins der Länge n geben.

Satz 1.1. Seien A und B Programmiersprachen. Es existiert eine Konstante $c_{A,B}$, die nur von A und B abhängig ist, so dass $\forall w \in \Sigma^*_{\text{Bool}}$

$$\left|K_{C_A}(w) - K_{C_B}\right| \leq c_{A,B}$$

Beweis 1.4. trivial

Def.! 1.21 (Zufällig)**.** Ein Wort $w \in \Sigma^*_{\text{Bool}}$ heißt zufällig, falls $K_C(w) \geq |w|$. Eine Zahl n heißt zufällig, falls

$$K_c(n) = K_c(\text{Bin}(n)) \geq \lceil \text{ld}(n+1) \rceil - 1$$

2 Endliche Automaten

Def.! 2.1 (**DEA!** (**DEA!**))**.** Ein **DEA!** ist ein Quintupel $M = (Q, \Sigma, \delta, q_0, F)$ aus

- einer endlichen Menge von Zuständen Q
- einem Eingabealphabet Σ
- einer Transitionsfunktion $\delta \colon Q \times \Sigma \to Q$
- einen Startzustand $q_0 \in Q$
- eine Menge von akzeptierenden Zuständen $F \subseteq Q$

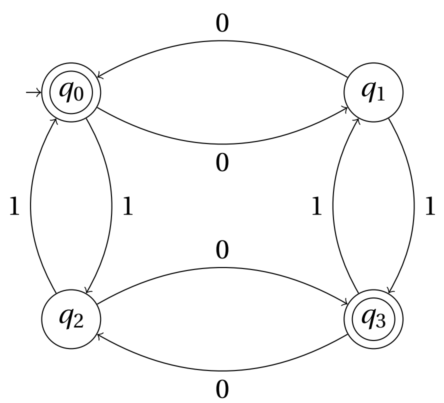
  Anmerkung: F kann leeres Element sein also kann der Automat auch keine Zustände annehmen!

Notation 2.1. Anstelle von $\delta(q, a) = q$ schreiben wir

$$p \xrightarrow{a}_M q$$

Beispiel:

M^*:



$M = \left(\{q_0, q_1, q_2, q_3\}, \Sigma_{\text{Bool}} m, \delta, q_0, \{q_0, q_3\}\right)$ mit δ:

	0	1
q_0	q_1	q_2
q_1	q_0	q_3
q_2	q_3	q_0
q_3	q_2	q_1

Def.! 2.2. Sei $M = (Q, \Sigma, \delta, q_0, F)$ ein **DEA!**. Eine Konfiguration von M ist ein Paar aus $Q \times \Sigma^*$. (q_0, w) heißt Startkonfiguration und (q, ϵ) Endkonfiguration (für **bel.!** $w \in \Sigma^*$). Gilt für eine Endkonfiguration (q, ϵ) das $q \in F$, so heißt diese akzeptierend, ansonsten verwerfend. □

Def.! 2.3. Sei $M = (Q, \Sigma, \delta, q_0, F)$ ein **DEA!**. Ein Konfigurationsübergang (ein Schritt) ist eine Relation $\vdash_M$: $(Q \times \Sigma^*) \times (Q \times \Sigma^*)$, die **def.!** ist durch

$$(p, \underbrace{a}_{\text{Lesekopf}} \overbrace{w}^{\text{Band}}) \vdash (q, w) \textbf{ gdw.! } \delta(p, a) = q$$

$$q \xrightarrow{a} q$$

mit $p, q \in Q$, $w \in \Sigma^*$ und $a \in \Sigma$. $\vdash_u$ nennen wir Schrittrelation.

Def.! 2.4. Sei $M = (Q, \Sigma, \delta, q_0, F)$ ein **DEA!** und $w \in \Sigma^*$. Eine Berechnung c von M ist eine endliche Folge von Konfigurationen

$$c = c_0, \ldots, c_n \text{ mit } c_i \vdash c_{i+1} (1 \leq i \leq n-1)$$

c ist eine Berechnung von M für eine Eingabe w, falls $c_0 = (q_0, w)$ (Startkonfiguration) und $c_n = (q, \epsilon)$ (Endkonfiguration). Falls c_n **akzept.!** (**akzept.!**) Endkonfiguration ist, so ist C **akzept.!** Berechnung; ansonsten verwerfende Berechnung.

Beispiel:

$$(q_0, \underline{1}001) \vdash (q_2, 001) \vdash (q_3, 01) \vdash (q_2, 1) \vdash (\underbrace{q_0}_{\in F}, \epsilon)$$

ist **akzept.!** Berechnung von M^* (zu GI(V)-15.04.2009-DIA-1) auf 1001

Def.! 2.5. Sei $M = (Q, \Sigma, \delta, q_0, F)$ ein **DEA!**. Wir definieren: $\vdash_M^*: (Q \times \Sigma^*) \times (Q \times \Sigma^*)$ als reflexiven und transitiven Abschluss von $\vdash$:

- $(q, w) \vdash^* (q, w)\ \forall q \in Q, w \in \Sigma^*$
- $(p, uv) \vdash^* (q, v)$, falls $u = a_1 \dots a_n (a_i \in \Sigma)\ \exists q_1, \dots, q_{n-1} \in Q$, so dass

  $$(p, a_1 \dots a_n v) \vdash (q_1, a_1 \dots a_n v) \vdash \dots \vdash (q_{n-1}, a_n v) \vdash (q, v)$$

  für $p, q \in Q\ u, v \in \Sigma^*$

Die Fortsetzung $\hat{\delta}: Q \times \Sigma^* \to Q$ der Transitionsfunktion δ auf Wörter **def.!** wir induktiv durch

- $\hat{\delta}(q, \epsilon) = q$
- $\hat{\delta}(q, wa) = \delta(\hat{\delta}(q, w))\ \forall w \in \Sigma^*, a \in \Sigma$ und $q \in Q$

Notation 2.2. Falls $\hat{\delta}(q, w) = p$, so schreiben wir

$$q \xrightarrow{w} p$$

und sagen es existiert ein Pfad von q nach p mit Beschriftung w.

Def.! 2.6. Sei $M = (Q, \Sigma, \delta, q_0, F)$ ein **DEA!**. Die von M akzeptierte Sprache $L(M)$ ergibt sich aus:

$$\begin{aligned} L(M) &= \{w \in \Sigma^* | \hat{\delta}(q_0, w) \in F\} \\ &= \left\{w \in \Sigma^* | q_0 \xrightarrow{w} q \text{ und } q \in F\right\} \\ &= \left\{w \in \Sigma^* | (q_0, w) \overset{*}{\vdash} (q, \epsilon) \text{ mit } q \in F\right\} \end{aligned}$$

Sprachen, die von einem **DEA!** erkannt werden nennt man reguläre Sprachen. Zwei **DEA!**s M_1 und M_2 sind äquivalent **gdw.!** $L(M_1) = L(M_2)$