8 ÜBERSETZUNGEN UND CODIERUNGEN

Von natürlichen Sprachen weiß man, dass man *übersetzen* kann. Beschränken wir uns im weiteren der Einfachheit halber als erstes auf Inschriften. Was ist dann eine Übersetzung? Das hat zwei Aspekte:

1. eine Zuordnung von Wörtern einer Sprache zu Wörtern einer anderen Sprache, die
2. die schöne Eigenschaft hat, dass jedes Ausgangswort und seine Übersetzung die gleiche Bedeutung haben.

Als erstes schauen wir uns ein einfaches Beispiel für Wörter und ihre Bedeutung an: verschiedene Methoden der Darstellung natürlicher Zahlen.

8.1 VON WÖRTERN ZU ZAHLEN UND ZURÜCK

8.1.1 Dezimaldarstellung von Zahlen

Wir sind gewohnt, natürliche Zahlen im sogenannten Dezimalsystem notieren, das aus Indien kommt (siehe auch Kapitel 12 über den Algorithmusbegriff):

- Verwendet werden die Ziffern des Alphabetes $Z_{10} = \{0, \ldots, 9\}$.
- Die Bedeutung $\mathrm{num}_{10}(x)$ einer einzelnen Ziffer x als Zahl ist durch die folgende Tabelle gegeben:

x	0	1	2	3	4	5	6	7	8	9
$\mathrm{num}_{10}(x)$	0	1	2	3	4	5	6	7	8	9

  Man beachte, dass in der ersten Zeile der Tabelle *Zeichen* stehen, und in der zweiten Zeile *Zahlen*. Genauer gesagt stehen in der ersten Zeile Zeichen, die für sich stehen, und in der zweiten Zeile Zeichen, die Sie als Zahlen interpretieren sollen.

  Also ist num_{10} eine Abbildung $\mathrm{num}_{10}\colon Z_{10} \to \mathbb{Z}_{10}$.

- Für die Bedeutung eines ganzen Wortes $x_{k-1}\cdots x_0 \in Z_{10}^*$ von Ziffern wollen wir $\mathrm{Num}_{10}(x_{k-1}\cdots x_0)$ schreiben. In der Schule hat man gelernt, dass das gleich

$$10^{k-1} \cdot \mathrm{num}_{10}(x_{k-1}) + \cdots + 10^1 \cdot \mathrm{num}_{10}(x_1) + 10^0 \cdot \mathrm{num}_{10}(x_0)$$

  ist. Wir wissen inzwischen, wie man die Pünktchen vermeidet. Und da

$$\begin{aligned} &10^{k-1} \cdot \mathrm{num}_{10}(x_{k-1}) + \cdots + 10^1 \cdot \mathrm{num}_{10}(x_1) + 10^0 \cdot \mathrm{num}_{10}(x_0) \\ &\quad = 10 \left(10^{k-2} \cdot \mathrm{num}_{10}(x_{k-1}) + \cdots + 10^0 \cdot \mathrm{num}_{10}(x_1) \right) + 10^0 \cdot \mathrm{num}_{10}(x_0) \end{aligned}$$

ist, definiert man $\mathrm{Num}_{10}\colon Z_{10}^* \to \mathbb{N}_0$ so:

$$\mathrm{Num}_{10}(\varepsilon) = 0$$
$$\text{für jedes } w \in Z_{10}^* \text{ für jedes } x \in Z_{10} : \mathrm{Num}_{10}(wx) = 10 \cdot \mathrm{Num}_{10}(w) + \mathrm{num}_{10}(x)$$

8.1.2 Andere unbeschränkte Zahldarstellungen

Gottfried Wilhelm Leibniz

Gottfried Wilhelm Leibniz wurde am 1. Juli 1646 in Leipzig geboren und starb am 14. November 1716 in Hannover. Er war Philosoph, Mathematiker, Physiker, Bibliothekar und vieles mehr. Leibniz baute zum Beispiel die erste Maschine, die zwei Zahlen multiplizieren konnte.
Leibniz hatte erkannt, dass man die natürlichen Zahlen nicht nur mit den Ziffern $0, \ldots, 9$ notieren kann, sondern dass dafür 0 und 1 genügen. Er hat dies in einem Brief vom 2. Januar 1697 an den Herzog von Braunschweig-Wolfenbüttel beschrieben (siehe auch `http://www.hs-augsburg.de/~harsch/germanica/Chronologie/17Jh/Leibniz/lei_bina.html`, 10.11.2015) und im Jahre 1703 in einer Zeitschrift veröffentlicht. Abbildung 8.1 zeigt Beispielrechnungen mit Zahlen in Binärdarstellung aus dieser Veröffentlichung.

Binärdarstellung

Bei der *Binärdarstellung* von nichtnegativen ganzen Zahlen geht man analog zur Dezimaldarstellung vor. Als Ziffernmenge benutzt man $Z_2 = \{0, 1\}$ und definiert

$$\mathrm{num}_2(0) = 0$$
$$\mathrm{num}_2(1) = 1$$
$$\mathrm{Num}_2(\varepsilon) = 0$$
$$\text{sowie} \quad \text{für jedes } w \in Z_2^* \text{ für jedes } x \in Z_2 : \mathrm{Num}_2(wx) = 2 \cdot \mathrm{Num}_2(w) + \mathrm{num}_2(x)$$

Damit ist dann z. B.

$$\begin{aligned}
\mathrm{Num}_2(1101) &= 2 \cdot \mathrm{Num}_2(110) + 1 \\
&= 2 \cdot (2 \cdot \mathrm{Num}_2(11) + 0) + 1 \\
&= 2 \cdot (2 \cdot (2 \cdot \mathrm{Num}_2(1) + 1) + 0) + 1 \\
&= 2 \cdot (2 \cdot (2 \cdot 1 + 1) + 0) + 1 \\
&= 2^3 \cdot 1 + 2^2 \cdot 1 + 2^1 \cdot 0 + 2^0 \cdot 1 \\
&= 13
\end{aligned}$$

 © worsch&wacker 2008–2015

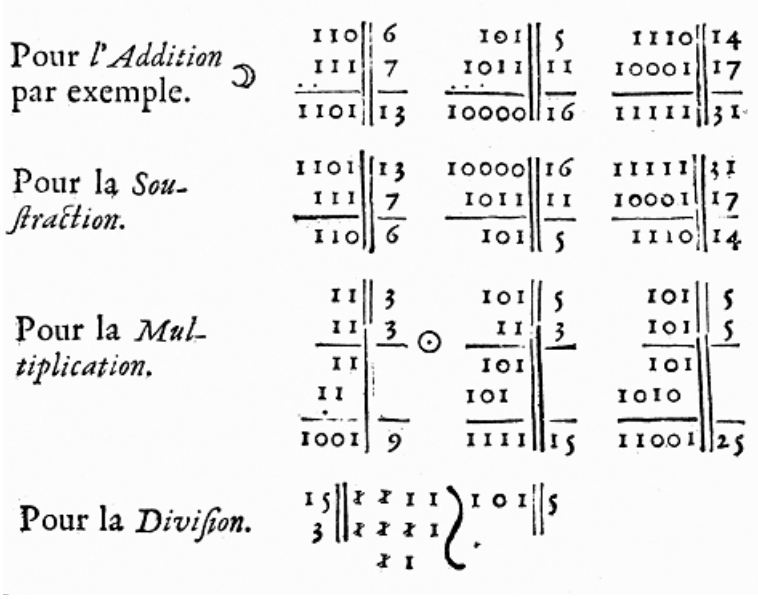


Abbildung 8.1: Ausschnitt aus dem Aufsatz „Explication de l'Arithmétique Binaire“ von Leibniz, Quelle: http://commons.wikimedia.org/wiki/Image:Leibniz_binary_system_1703.png (10.11.2015)

Bei der *Hexadezimaldarstellung* oder Sedezimaldarstellung benutzt man 16 Ziffern des Alphabets $Z_{16} = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F\}$. (Manchmal werden statt der Großbuchstaben auch Kleinbuchstaben verwendet.) *Hexadezimaldarstellung*

x	0	1	2	3	4	5	6	7
$\mathrm{num}_{16}(x)$	0	1	2	3	4	5	6	7
x	8	9	A	B	C	D	E	F
$\mathrm{num}_{16}(x)$	8	9	10	11	12	13	14	15

Die Zuordnung von Wörtern zu Zahlen ist im Wesentlichen gegeben durch

$$\text{für jedes } w \in Z_{16}^* \text{ für jedes } x \in Z_{16} : \mathrm{Num}_{16}(wx) = 16 \cdot \mathrm{Num}_{16}(w) + \mathrm{num}_{16}(x)$$

Ein Problem ergibt sich dadurch, dass die Alphabete Z_2, Z_3, usw. nicht disjunkt sind. Daher ist z. B. die Zeichenfolge 111 mehrdeutig: Um sagen zu können, welche Zahl hier repräsentiert wird, muss man wissen, zu welcher Basis die Zahl dargestellt ist. Zum Beispiel ist

 © worsch&wacker 2008–2015

$\mathrm{Num}_2(111)$	die Zahl sieben,
$\mathrm{Num}_8(111)$	die Zahl dreiundsiebzig,
$\mathrm{Num}_{10}(111)$	die Zahl einhundertelf und
$\mathrm{Num}_{16}(111)$	die Zahl zweihundertdreiundsiebzig.

Deswegen ist es in manchen Programmiersprachen so, dass für Zahldarstellungen zu den Basen 2, 8 und 16 als Präfix respektive `0b`, `0o` und `0x` vorgeschrieben sind. Dann sieht man der Darstellung unmittelbar an, wie sie zu interpretieren ist. In anderen Programmiersprachen sind entsprechende Darstellungen gar nicht möglich.

8.1.3 Ganzzahlige Division mit Rest

So wie man zu einem Wort die dadurch repräsentierte Zahl berechnen kann, kann man auch umgekehrt zu einer nichtnegativen ganzen Zahl $n \in \mathbb{N}_0$ eine sogenannte k-äre Darstellung berechnen.

Um das einzusehen betrachten wir als Vorbereitung zwei binäre Operationen, die Ihnen im Laufe dieser und anderer Vorlesungen noch öfter über den Weg laufen werden. Sie heißen **div** und **mod** und sind wie folgt definiert: Die Operation **mod** liefert für zwei Argumente $x \in \mathbb{N}_0$ und $y \in \mathbb{N}_+$ als Funktionswert x **mod** y den Rest der ganzzahligen Division von x durch y. Und die Operation **div** liefere für $x \in \mathbb{N}_0$ und $y \in \mathbb{N}_+$ als Funktionswert x **div** y den Wert der ganzzahligen Division von x durch y. Mit anderen Worten gilt stets:

div
mod

$$x = y \cdot (x \textbf{ div } y) + (x \textbf{ mod } y) \qquad \text{und} \qquad 0 \leq (x \textbf{ mod } y) < y \tag{8.1}$$

Etwas anders ausgedrückt gilt das folgende: Wenn eine Zahl $x \in \mathbb{N}_0$ für ein $k \in \mathbb{N}_+$ in der Form $x = k \cdot p_x + r_x$ dargestellt ist, wobei $p_x \in \mathbb{N}_0$ und $0 \leq r_x < k$, also $r_x \in \mathbb{Z}_k$, ist, dann ist gerade $p_x = x$ **div** k und $r_x = x$ **mod** k.

Hieraus ergibt sich schnell folgendes Lemma.

8.1 Lemma. Für jede $x, y \in \mathbb{N}_0$ und jedes $k \in \mathbb{N}_+$ gilt:

$$(x + y) \textbf{ mod } k = ((x \textbf{ mod } k) + (y \textbf{ mod } k)) \textbf{ mod } k$$

8.2 Beweis. Es sei $x = k \cdot p_x + r_x$ und $y = k \cdot p_y + r_y$ mit $r_x, r_y \in \mathbb{Z}_k$. Dann ist

$$\begin{aligned}(x + y) \textbf{ mod } k &= (kp_x + r_x + kp_y + r_y) \textbf{ mod } k \\ &= (r_x + r_y) \textbf{ mod } k \\ &= ((x \textbf{ mod } k) + (y \textbf{ mod } k)) \textbf{ mod } k\end{aligned}$$

■

8.1.4 Von Zahlen zu ihren Darstellungen

Es sei nun $k \geq 2$.

Mit Hilfe der beiden eben eingeführten Funktionen kann man dann eine sogenannte *k-äre Darstellung* oder auch *k-äre Repräsentation* einer Zahl festlegen. Dazu benutzt man ein Alphabet Z_k mit k Ziffern, deren Bedeutungen die Zahlen in $\mathbb{Z}_k$ sind. Für $i \in \mathbb{Z}_k$ sei $\text{repr}_k(i)$ dieses Zeichen. Die Abbildung repr_k ist also gerade die Umkehrfunktion zu num_k.

k-äre Darstellung
k-äre Repräsentation

Gesucht ist nun eine Repräsentation von $n \in \mathbb{N}_0$ als Wort $w \in Z_k^*$ mit der Eigenschaft $\text{Num}_k(w) = n$. Dabei nimmt man die naheliegende Definition von Num_k an.

Wie wir gleich sehen werden, gibt es immer solche Wörter. Und es sind dann auch immer gleich unendlich viele, denn wenn $\text{Num}_k(w) = n$ ist, dann auch $\text{Num}_k(0w) = n$ (einfach nachrechnen).

Die Funktion Repr_k definieren wir so:

$$\text{Repr}_k(n) = \begin{cases} \text{repr}_k(n) & \text{falls } n < k \\ \text{Repr}_k(n \textbf{ div } k) \cdot \text{repr}_k(n \textbf{ mod } k) & \text{falls } n \geq k \end{cases} \tag{8.2}$$

Dabei bedeutet der Punkt $\cdot$ in der zweiten Zeile Konkatenation. Hier liegt wieder einmal ein induktive Definition vor, aber es ist nicht so klar wie bisher, dass tatsächlich für jedes $n \in \mathbb{N}_0$ ein Funktionswert $\text{Repr}_k(n)$ definiert wird. Daher wollen wir das beweisen. Genauer betrachten wir für $m \in \mathbb{N}_+$ die Aussagen $\mathcal{A}_m$ der Form:

„Für alle $n \in \mathbb{N}_n$ mit $n < k^m$ ist $\text{Repr}_k(n)$ definiert."

oder anders aber äquivalent formuliert:

„Für alle $n \in \mathbb{N}_n$ gilt: Wenn $n < k^m$ ist, dann ist $\text{Repr}_k(n)$ definiert."

Im *Induktionsanfang* müssen wir Aussage $\mathcal{A}_1$ beweisen, die besagt, dass $\text{Repr}_k(n)$ für alle $n \in \mathbb{N}_0$ mit $n < k$ definiert ist. Das ist aber klar, denn dann ist laut (8.2) gerade $\text{Repr}_k(n) = \text{repr}_k(n)$.

Für den Induktionsschritt sei nun $m \in \mathbb{N}_+$. Als *Induktionsvoraussetzung* können wir benutzen:

„Für alle $n \in \mathbb{N}_n$ mit $n < k^m$ ist $\text{Repr}_k(n)$ definiert."

Zu zeigen ist, dass dann auch gilt:

„Für alle $n \in \mathbb{N}_n$ mit $n < k^{m+1}$ ist $\text{Repr}_k(n)$ definiert."

Sei also $n \in \mathbb{N}_0$ und $n < k^{m+1}$. Dann sind zwei Fälle möglich:

- Es ist sogar $n < k^m$. Dann ist $\text{Repr}_k(n)$ gemäß der Induktionsvoraussetzung definiert.
- Es ist $k^m \leq n < k^{m+1}$. Dann ist $n \textbf{ div } k < k^m$ und somit nach Induktionsvoraussetzung $\text{Repr}_k(n \textbf{ div } k)$ definiert. Folgich ist aber wieder nach (8.2) auch $\text{Repr}_k(n) = \text{Repr}_k(n \textbf{ div } k) \cdot \text{repr}_k(n \textbf{ mod } k)$ definiert.

 © worsch&wacker 2008–2015

Es sei noch angemerkt, dass mit Ausnahme der 0 die Funktion Repr_k zu gegebenem $n \in \mathbb{N}_0$ das (es ist eindeutig) kürzeste Wort $w \in Z_k^*$ mit $\mathrm{Num}_k(w) = n$. Es ist also stets

$$\mathrm{Num}_k(\mathrm{Repr}_k(n)) = n\ .$$

Man beachte, dass umgekehrt $\mathrm{Repr}_k(\mathrm{Num}_k(w))$ im allgemeinen nicht unbedingt wieder w ist, weil „überflüssige" führende Nullen wegfallen.

8.1.5 Beschränkte Zahlbereiche und Zahldarstellungen

In Kapitel 10 werden Sie ein erstes Modell für Prozessoren kennenlernen, die (mikroprammierte) *Minimalmaschine* Mima. Sie ist nicht wirklich minimal, aber an einigen Stellen wird sie im Vergleich zu realen Prozessoren wie Sie in Desktop-Rechnern, Mobiltelefonen und anderen Geräten verbaut sind, stark vereinfacht sein. Wichtige Aspekte werden aber auch schon an der Mima sichtbar werden.

Dazu gehört, dass man nicht beliebig große Zahlen repräsentieren kann. Einzelne Zahlen (und auch anders zu interpretierende Objekte) werden in sogenannten *Registern* gespeichert werden, und zwar in der Mima immer als Bitwörter der Länge 24. In einem Register lassen sich also maximal 2^{24} verschiedene Zahlen speichern. Die Frage ist: welche und in welcher Repräsentation?

Eine einfache (aber nicht zutreffende) Vorgehensweise wäre diese:

- Für $\ell \in \mathbb{N}_+$ sei die Funktion $\mathrm{bin}_\ell\colon \mathbb{Z}_{2^\ell} \to \{0,1\}^\ell$ definiert vermöge der Festlegung
$$\mathrm{bin}_\ell(n) = 0^{\ell - |\mathrm{Repr}_2(n)|}\mathrm{Repr}_2(n)$$
Es ist also stets $|\mathrm{bin}_\ell(n)| = \ell$ und $\mathrm{Num}_2(\mathrm{bin}_\ell(n)) = n$.
- Dann könnte man in der Mima jede Zahl $n \in \mathbb{Z}_{2^{24}}$ in der Form $\mathrm{bin}_{24}(n)$ darstellen.

Aber „natürlich" möchte man auch negative Zahlen repräsentieren können. Und es wäre praktisch, wenn man für deren Verarbeitung keine extra Recheneinheiten benötigen würde, sondern z. B. ein sogenanntes Addierwerk immer das Gewünschte leistet, gleich, ob die Argumente für negative oder für nichtnegative Zahlen stehen.

Zweierkomplement-Darstellung

Es zeigt sich, dass man das tatsächlich erreichen kann. Im Folgenden stellen wir eine solche Repräsentation vor, die sogenannte *Zweierkomplement-Darstellung*.

Auch dafür muss man als erstes eine *feste Länge* der Zahldarstellungen festlegen. Wir bezeichnen Sie mit ℓ. Es ist $\ell \in \mathbb{N}_+$ und sinnvollerweise $\ell \geq 2$. Die Menge der Zahlen, die man in Zweierkomplementdarstellung der Länge ℓ repräsentieren kann ist

$$\mathbb{K}_\ell = \{x \in \mathbb{Z} \mid -2^{\ell-1} \leq x \leq 2^{\ell-1} - 1\}\ .$$

(Dabei schreiben wir „$a \leq b \leq c$" als Abkürzung für „$a \leq b$ und $b \leq c$".) Es sind also

z. B.

$$\mathbb{K}_2 = \{-2, -1, 0, 1\}$$
$$\text{und} \qquad \mathbb{K}_8 = \{-128, -127, \ldots, -1, 0, 1, \ldots, 127\}$$

Die Zweierkomplementdarstellung $\text{Zkpl}_\ell : \mathbb{K}_\ell \to \{0,1\}^\ell$ der Länge ℓ ist für alle $x \in \mathbb{K}_\ell$ wie folgt definiert:

$$\text{Zkpl}_\ell(x) = \begin{cases} 0\text{bin}_{\ell-1}(x) & \text{falls } x \geq 0 \\ 1\text{bin}_{\ell-1}(2^{\ell-1} + x) & \text{falls } x < 0 \end{cases}$$

Man kann sich überlegen, dass auch gilt:

$$\text{Zkpl}_\ell(x) = \begin{cases} \text{bin}_\ell(x) & \text{falls } x \geq 0 \\ \text{bin}_\ell(2^\ell + x) & \text{falls } x < 0 \end{cases}$$

Dass diese Zahlendarstellung Eigenschaften hat, die z. B. die technische Implementierung eines Addierwerkes erleichtern, werden Sie (hoffentlich) an anderer Stelle lernen.

8.2 VON EINEM ALPHABET ZUM ANDEREN

8.2.1 Ein Beispiel: Übersetzung von Zahldarstellungen

Wir betrachten die Funktion $\text{Trans}_{2,16} = \text{Repr}_2 \circ \text{Num}_{16}$ von Z_{16}^* nach Z_2^*. Sie bildet zum Beispiel das Wort A3 ab auf

$$\text{Repr}_2(\text{Num}_{16}(\texttt{A3})) = \text{Repr}_2(163) = \texttt{10100011}\ .$$

Der wesentliche Punkt ist, dass die beiden Wörter A3 und 10100011 in der jeweils „naheliegenden“ Interpretation die gleiche Bedeutung haben: die Zahl einhundertdreiundsechzig.

Allgemein wollen wir folgende Sprechweisen vereinbaren. Sehr oft, wie zum Beispiel gesehen bei Zahldarstellungen, schreibt man Wörter einer formalen Sprache L über einem Alphabet und meint aber etwas anderes, ihre Bedeutung. Die Menge der Bedeutungen der Wörter aus L ist je nach Anwendungsfall sehr unterschiedlich. Es kann so etwas einfaches sein wie Zahlen, oder so etwas kompliziertes wie die Bedeutung der Ausführung eines Java-Programmes. Für so eine Menge von „Bedeutungen“ schreiben wir im folgenden einfach Sem.

Wir gehen nun davon aus, dass zwei Alphabete A und B gegeben sind, und zwei Abbildungen $\text{sem}_A : L_A \to \text{Sem}$ und $\text{sem}_B : L_B \to \text{Sem}$ von formalen Sprachen $L_A \subseteq A^*$ und $L_B \subseteq B^*$ in die *gleiche* Menge Sem von Bedeutungen.

Übersetzung

Eine Abbildung $f : L_A \to L_B$ heiße eine *Übersetzung* bezüglich sem_A und sem_B, wenn f die Bedeutung erhält, d. h.

$$\text{für jedes } w \in L_A : \text{sem}_A(w) = \text{sem}_B(f(w))$$

Betrachten wir noch einmal die Funktion $\text{Trans}_{2,16} = \text{Repr}_2 \circ \text{Num}_{16}$. Hier haben wir den einfachen Fall, dass $L_A = A^* = Z_{16}^*$ und $L_B = B^* = Z_2^*$. Die Bedeutungsfunktionen sind $\text{sem}_A = \text{Num}_{16}$ und $\text{sem}_B = \text{Num}_2$. Dass bezüglich dieser Abbildungen $\text{Trans}_{2,16}$ tatsächlich um eine Übersetzung handelt, kann man leicht nachrechnen:

$$\begin{aligned}
\text{sem}_B(f(w)) &= \text{Num}_2(\text{Trans}_{2,16}(w)) \\
&= \text{Num}_2(\text{Repr}_2(\text{Num}_{16}(w))) \\
&= \text{Num}_{16}(w) \\
&= \text{sem}_A(w)
\end{aligned}$$

Im allgemeinen kann die Menge der Bedeutungen recht kompliziert sein. Wenn es um die Übersetzung von Programmen aus einer Programmiersprache in eine andere Programmiersprache geht, dann ist die Menge Sem die Menge der Bedeutungen von Programmen. Als kleine Andeutung wollen hier nur erwähnen, dass man dann z. B. die Semantik einer Zuweisung $x \leftarrow 5$ definieren könnte als die Abbdildung, die aus einer Speicherbelegung m die Speicherbelegung $\text{memwrite}(m, x, 5)$ macht (siehe Kapitel 9). Eine grundlegende Einführung in solche Fragestellungen können Sie in Vorlesungen über die Semantik von Programmiersprachen bekommen.

Warum macht man Übersetzungen? Zumindest die folgenden Möglichkeiten fallen einem ein:

- *Lesbarkeit:* Übersetzungen können zu kürzeren und daher besser lesbaren Texten führen. `A3` ist leichter erfassbar als `10100011` (findet der Autor dieser Zeilen).
- *Verschlüsselung:* Im Gegensatz zu verbesserter Lesbarkeit übersetzt man mitunter gerade deshalb, um die Lesbarkeit möglichst unmöglich zu machen, jedenfalls für Außenstehende. Wie man das macht, ist Gegenstand von Vorlesungen über Kryptographie.
- *Kompression:* Manchmal führen Übersetzungen zu kürzeren Texten, die also weniger Platz benötigen. Und zwar *ohne* zu einem größeren Alphabet überzugehen. Wir werden im Abschnitt 8.3 über Huffman-Codes sehen, warum und wie das manchmal möglich ist.
- *Fehlererkennung* und *Fehlerkorrektur:* Manchmal kann man Texte durch Übersetzung auf eine Art länger machen, dass man selbst dann, wenn ein korrekter Funktionswert $f(w)$ „zufällig“ „kaputt“ gemacht wird (z. B. durch Übertragungsfehler auf einer Leitung) und nicht zu viele Fehler passieren, man

die Fehler korrigieren kann, oder zumindest erkennt, dass Fehler passiert sind. Ein typisches Beispiel sind lineare Codes, von denen Sie (vielleicht) in anderen Vorlesung hören werden.

Es gibt einen öfter anzutreffenden Spezialfall, in dem man sich um die Einhaltung der Forderung $\text{sem}_A(w) = \text{sem}_B(f(w))$ keine Gedanken machen muss. Zumindest bei Verschlüsselung, aber auch bei manchen Anwendungen von Kompression ist es so, dass man vom Übersetzten $f(x)$ eindeutig zurückkommen können möchte zum ursprünglichen x. Dann ist f mit anderen Worten injektiv. In diesem Fall kann man die Bedeutung sem_B im wesentlichen *definieren* durch die Festlegung $\text{sem}_B(f(x)) = \text{sem}_A(x)$. Man mache sich klar, dass an dieser Stelle die Injektivität von f wichtig ist, damit sem_B wohldefiniert ist. Denn wäre für zwei $x \neq y$ zwar $\text{sem}_A(x) \neq \text{sem}_A(y)$ aber die Funktionswerte $f(x) = f(y) = z$, dann wäre nicht klar, was $\text{sem}_B(z)$ sein soll.

Wenn eine Übersetzung f injektiv ist, wollen wir das eine *Codierung* nennen. Für $w \in L_A$ heißt $f(w)$ ein *Codewort* und die Menge $\{f(w) \mid w \in L_A\}$ aller Codewörter heißt dann auch ein *Code*.

Codierung
Codewort
Code

Es stellt sich die Frage, wie man eine Übersetzung vollständig spezifiziert. Man kann ja nicht für im allgemeinen unendliche viele Wörter $w \in L_A$ einzeln erschöpfend aufzählen, was $f(w)$ ist.

Eine Möglichkeit bieten sogenannte Homomorphismen und Block-Codierungen, auf die wir im Folgenden noch genauer eingehen werden.

8.2.2 Homomorphismen

Es seien A und B zwei Alphabete und $h : A^* \to B^*$ eine Abbildung. Eine solche Abbildung h nennt man einen *Homomorphismus*, wenn für jedes $w_1 \in A^*$ und jedes $w_2 \in A^*$ gilt:

Homomorphismus

$$h(w_1 w_2) = h(w_1)h(w_2) \ .$$

Ein Homomorphismus heißt *ε-frei*, wenn für alle $x \in A$ gilt: $h(x) \neq \varepsilon$.

ε-freier Homomorphismus

Für jeden Homomorphismus h gilt $h(\varepsilon) = h(\varepsilon\varepsilon) = h(\varepsilon)h(\varepsilon)$. Also ist $|h(\varepsilon)| = |h(\varepsilon)| + |h(\varepsilon)|$, d. h. $|h(\varepsilon)| = 0$, und folglich muss für jeden Homomorphismus $h(\varepsilon) = \varepsilon$ sein.

Homomorphismen sind schon eindeutig festgelegt, wenn man das Bild jedes einzelnen Symboles kennt:

8.3 Lemma. Es seien A und B zwei Alphabete und $h : A^* \to B^*$ und $g : A^* \to B^*$ zwei Homomorphismen. Dann gilt: Wenn für jedes $x \in A$ gilt, dass $h(x) = g(x)$ ist, dann gilt sogar schon für jedes $w \in \mathcal{A}^*$, dass $h(w) = g(w)$ ist.

8.4 Beweis. Es seien A, B, h und g wie in den Voraussetzungen des Lemmas und es gelte für jedes $x \in A$ gilt, dass $h(x) = g(x)$ ist. Es ist zu zeigen, dass für jedes $w \in A^*$ gilt: $h(w) = g(w)$. Ein möglicher Beweis benutzt Induktion über die Wortlänge.

Der *Induktionsanfang* ($w = \varepsilon$) ist leicht, da nach der Überlegung vor dem Lemma stets $h(\varepsilon) = \varepsilon = g(\varepsilon)$ ist.

Für den *Induktionsschritt* seien $w \in A^*$ und $x \in A$ und es gelte die *Induktionsvoraussetzung*: $h(w) = g(w)$. Dann gilt auch:

$$\begin{aligned} h(wx) &= h(w)h(x) && \text{da } h \text{ Homomorphismus} \\ &= g(w)h(x) && \text{Induktionsvoraussetzung} \\ &= g(w)g(x) && \text{Voraussetzung des Lemmas} \\ &= g(wx) && \text{da } g \text{ Homomorphismus} \end{aligned}$$

■

Wenn ein Homomorphismus h vorliegt, dann ist er also durch die Bilder $h(x)$ der einzelnen Symbole eindeutig festgelegt. Und tatsächlich legt auch jede Abbildung $f : A \to B^*$ schon einen Homomorphimus fest. Dazu definiert man eine Abbildung $f^{**} : A^* \to B^*$ vermöge

$$\begin{gathered} f^{**}(\varepsilon) = \varepsilon \\ \text{für jedes } w \in A^*, \text{für jedes } x \in A : f^{**}(wx) = f^{**}(w)f(x) \end{gathered}$$

Dann gilt:

8.5 Lemma. Für jede Abbildung $f : A \to B^*$ ist f^{**} ein Homomorphismus.

Den Beweis würde man wieder mittels vollständiger Induktion führen. Wir ersparen uns das an dieser Stelle.

Wann ein Homomorphismus $h : A^* \to B^*$ eine Codierung, also injektiv ist, ist im allgemeinen nicht ganz einfach zu sehen. Es gibt aber einen Spezialfall, in dem
präfixfrei das klar ist, nämlich dann, wenn h *präfixfrei* ist. Das bedeutet, dass für *keine* zwei verschiedenen Symbole $x_1, x_2 \in A$ gilt: $h(x_1)$ ist ein Präfix von $h(x_2)$.

Die Decodierung ist in diesem Fall relativ einfach. Allerdings hat man in vielen Fällen das Problem zu beachten, dass nicht alle Wörter aus B^* ein Codewort sind. Mit anderen Worten ist h im allgemeinen nicht surjektiv. Um die Decodierung trotzdem als totale Abbildung u definieren zu können, wollen wir hier als erstes festlegen, dass es sich um eine Abbildung $u : B^* \to (A \cup \{\bot\})^*$ handelt. Das zusätzliche Symbol $\bot$ wollen wir benutzen, wenn ein $w \in B^*$ gar kein Codewort ist und nicht decodiert werden kann. In diesem Fall soll $u(w)$ das Symbol $\bot$ enthalten.

Als Beispiel betrachten wir den Homomorphismus $h : \{\mathtt{a},\mathtt{b},\mathtt{c}\}^* \to \{\mathtt{0},\mathtt{1}\}^*$ mit $h(\mathtt{a}) = \mathtt{1}$, $h(\mathtt{b}) = \mathtt{01}$ und $h(\mathtt{c}) = \mathtt{001}$. Dieser Homomorphismus ist präfixfrei.

Wir schreiben nun zunächst einmal Folgendes hin:

$$u(w) = \begin{cases} \varepsilon, & \text{falls } w = \varepsilon \\ \mathtt{a}u(w'), & \text{falls } w = \mathtt{1}w' \\ \mathtt{b}u(w'), & \text{falls } w = \mathtt{01}w' \\ \mathtt{c}u(w'), & \text{falls } w = \mathtt{001}w' \\ \bot, & \text{sonst} \end{cases}$$

Sei w das Beispielcodewort $w = \mathtt{100101} = h(\mathtt{acb})$. Versuchen wir nachzurechnen, was die Abbildung u mit w „macht“:

$$\begin{aligned} u(\mathtt{100101}) &= \mathtt{a}u(\mathtt{00101}) \\ &= \mathtt{ac}u(\mathtt{01}) \\ &= \mathtt{acb}u(\varepsilon) \\ &= \mathtt{acb} \end{aligned}$$

Prima, das hat geklappt. Aber warum? In jedem Schritt war klar, welche Zeile der Definition von u anzuwenden war. Und warum war das klar? Im Wesentlichen deswegen, weil ein Wort w nicht gleichzeitig mit den Codierungen verschiedener Symbole anfangen kann; kein $h(x)$ ist Anfangsstück eines $h(y)$ für *verschiedene* $x, y \in A$. Das ist nichts anderes als die Präfixfreiheit von h.

Man spricht hier auch davon, dass die oben festgelegte Abbildung u *wohldefiniert* ist. Über Wohldefiniertheit muss man immer dann nachdenken, wenn ein Funktionswert potenziell „auf mehreren Wegen“ festgelegt wird. Dann muss man sich entweder klar machen, dass in Wirklichkeit wie im obigen Beispiel immer nur ein Weg „gangbar“ ist, oder dass auf den verschiedenen Wegen am Ende der gleiche Funktionswert herauskommt. Für diesen zweiten Fall werden wir später noch Beispiele sehen, z. B. in dem Kapitel über Äquivalenz- und Kongruenzrelationen.

wohldefiniert

Allgemein kann man bei einem präfixfreien Code also so decodieren:

$$u(w) = \begin{cases} \varepsilon, & \text{falls } w = \varepsilon \\ x\,u(w'), & \text{falls } w = h(x)w' \text{ für ein } x \in A \\ \bot, & \text{sonst} \end{cases}$$

Man beachte, dass das *nicht* heißt, dass man nur präfixfreie Codes decodieren kann. Es heißt nur, dass man nur präfixfreie Codes „so einfach“ decodieren kann.

 © worsch&wacker 2008–2015

Das praktische Beispiel schlechthin für einen Homomorphismus ist die Repräsentation des ASCII-Zeichensatzes (siehe Abschnitt 3.2.1) im Rechner. Das geht einfach so: Ein Zeichen x mit der Nummer n im ASCII-Code wird codiert durch dasjenige Wort $w \in \{0,1\}^8$ (ein sogenanntes Byte), für das $\mathrm{Num}_2(w) = n$ ist. Und längere Texte werden übersetzt, indem man nacheinander jedes Zeichen einzeln so abbildet.

Dieser Homomorphismus hat sogar die Eigenschaft, dass alle Zeichen durch Wörter gleicher Länge codiert werden. Das muss aber im allgemeinen nicht so sein. Im nachfolgenden Unterabschnitt kommen wir kurz auf einen wichtigen Fall zu sprechen, bei dem das nicht so ist.

8.2.3 Beispiel Unicode: UTF-8 Codierung

Auch die Zeichen des Unicode-Zeichensatz kann man natürlich im Rechner speichern. Eine einfache Möglichkeit besteht darin, analog zu ASCII für alle Zeichen die jeweiligen Nummern (Code Points) als jeweils gleich lange Wörter darzustellen. Da es so viele Zeichen sind (und (unter anderem deswegen) manche Code Points große Zahlen sind), bräuchte man für jedes Zeichen vier Bytes.

Nun ist es aber so, dass zum Beispiel ein deutscher Text nur sehr wenige Zeichen benutzen wird, und diese haben auch noch kleine Nummern. Man kann daher auf die Idee kommen, nach platzsparenderen Codierungen zu suchen. Eine von ihnen ist *UTF-8*.

UTF-8

Nachfolgend ist die Definition dieses Homomorphismus in Ausschnitten wiedergegeben. Sie stammen aus *RFC 3629* (http://tools.ietf.org/html/rfc3629, 10.11.2015).

RFC 3629

> The table below summarizes the format of these different octet types. The letter x indicates bits available for encoding bits of the character number.
>
> | Char. number range (hexadecimal) | UTF-8 octet sequence (binary) |
> |---|---|
> | 0000 0000 - 0000 007F | 0*xxxxxxx* |
> | 0000 0080 - 0000 07FF | 110*xxxxx* 10*xxxxxx* |
> | 0000 0800 - 0000 FFFF | 1110*xxxx* 10*xxxxxx* 10*xxxxxx* |
> | 0001 0000 - 0010 FFFF | 11110*xxx* 10*xxxxxx* 10*xxxxxx* 10*xxxxxx* |
>
> Encoding a character to UTF-8 proceeds as follows:

- Determine the number of octets required from the character number and the first column of the table above. It is important to note that the rows of the table are mutually exclusive, i.e., there is only one valid way to encode a given character.
- Prepare the high-order bits of the octets as per the second column of the table.
- Fill in the bits marked x from the bits of the character number, expressed in binary. Start by putting the lowest-order bit of the character number in the lowest-order position of the last octet of the sequence, then put the next higher-order bit of the character number in the next higher-order position of that octet, etc. When the x bits of the last octet are filled in, move on to the next to last octet, then to the preceding one, etc. until all x bits are filled in.

Da die druckbaren Zeichen aus dem ASCII-Zeichensatz dort die gleichen Nummern haben wie bei Unicode, hat dieser Homomorphismus die Eigenschaft, dass Texte, die nur ASCII-Zeichen enthalten, bei der ASCII-Codierung und bei UTF-8 die gleiche Codierung besitzen.

8.3 HUFFMAN-CODIERUNG

Es sei ein Alphabet A und ein Wort $w \in A^*$ gegeben. Eine sogenannte *Huffman-Codierung* von w ist der Funktionswert $h(w)$ einer Abbildung $h : A^* \to Z_2^*$, die ε-freier Homomorphismus ist. Die Konstruktion von h wird dabei „auf w zugeschnitten“, damit $h(w)$ besonders „schön“, d. h. in diesem Zusammenhang besonders kurz, ist.

Huffman-Codierung

Jedes Symbol $x \in A$ kommt mit einer gewissen absoluten Häufigkeit $N_x(w)$ in w vor. Der wesentliche Punkt ist, dass Huffman-Codes häufigere Symbole durch kürzere Wörter codieren und seltener vorkommende Symbole durch längere.

Wir beschreiben als erstes, wie man die $h(x)$ bestimmt. Anschließend führen wir interessante und wichtige Eigenschaften von Huffman-Codierungen auf, die auch der Grund dafür sind, dass sie Bestandteil vieler Kompressionsverfahren sind. Zum Schluß erwähnen wir eine naheliegende Verallgemeinerung des Verfahrens.

8.3.1 Algorithmus zur Berechnung von Huffman-Codes

Gegeben sei ein $w \in A^*$ und folglich die Anzahlen $N_x(w)$ aller Symbole $x \in A$ in w. Da man Symbole, die in w überhaupt nicht vorkommen, auch nicht codieren

muss, beschränken wir uns bei der folgenden Beschreibung auf den Fall, dass alle $N_x(w) > 0$ sind (w also eine surjektive Abbildung auf A ist).

Der Algorithmus zur Bestimmung eines Huffman-Codes arbeitet in zwei Phasen:

1. Zunächst konstruiert er Schritt für Schritt einen Baum. Die Blätter des Baumes entsprechen $x \in A$, innere Knoten, d. h. Nicht-Blätter entsprechen Mengen von Symbolen. Um Einheitlichkeit zu haben, wollen wir sagen, dass ein Blatt für eine Menge $\{x\}$ steht.

   An *jedem* Knoten wird eine Häufigkeit notiert. Steht ein Knoten für eine Knotenmenge $X \subseteq A$, dann wird als Häufigkeit gerade die Summe $\sum_{x \in X} N_x(w)$ der Häufigkeiten der Symbole in X aufgeschrieben. Bei einem Blatt ist das also einfach ein $N_x(w)$. Zusätzlich wird bei jedem Blatt das zugehörige Symbol x notiert.

   In dem konstruierten Baum hat jeder innere Knoten zwei Nachfolger, einen linken und einen rechten.
2. In der zweiten Phase werden alle Kanten des Baumes beschriftet, und zwar jede linke Kante mit `0` und jede rechte Kante mit `1`.

   Um die Codierung eines Zeichens x zu berechnen, geht man dann auf dem kürzesten Weg von der Wurzel des Baumes zu dem Blatt, das x entspricht, und konkateniert der Reihe nach alle Symbole, mit denen die Kanten auf diesem Weg beschriftet sind.

Wenn zum Beispiel das Wort w = `afebfecaffdeddccefbeff` gegeben ist, dann kann sich der Baum ergeben, der in Abbildung 8.2 dargestellt ist. In diesem Fall ist dann der Homomorphismus gegeben durch

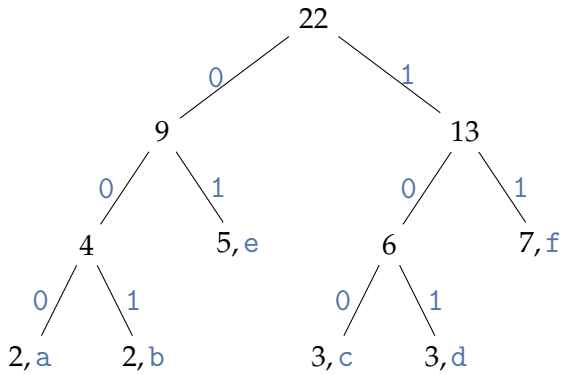


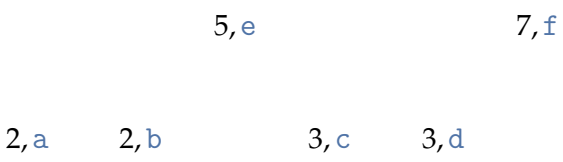
Abbildung 8.2: Ein Beispielbaum für die Berechnung eines Huffman-Codes.

x	a	b	c	d	e	f
h(x)	000	001	100	101	01	11

Es bleibt zu beschreiben, wie man den Baum konstruiert. Zu jedem Zeitpunkt hat man eine Menge M von „noch zu betrachtenden Symbolmengen mit ihren Häufigkeiten“. Diese Menge ist initial die Menge aller $\{x\}$ für $x \in A$ mit den zugehörigen Symbolhäufigkeiten, die wir so aufschreiben wollen:

$$M_0 = \{\ (2,\{\texttt{a}\})\ ,\ (2,\{\texttt{b}\})\ ,\ (3,\{\texttt{c}\})\ ,\ (3,\{\texttt{d}\})\ ,\ (5,\{\texttt{e}\})\ ,\ (7,\{\texttt{f}\})\ \}$$

Als Anfang für die Konstruktion des Baumes zeichnet man für jedes Symbol einen Knoten mit Markierung $(x, N_x(w))$. Im Beispiel ergibt sich



Solange eine Menge M_i noch mindestens zwei Paare enthält, tut man folgendes:

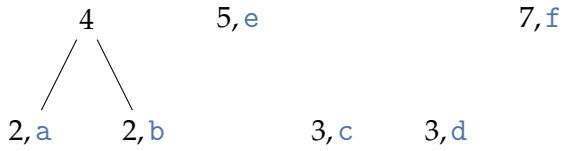
- Man bestimmt man eine Menge M_{i+1} wie folgt:
  - Man wählt zwei Paare (X_1, k_1) und (X_2, k_2), deren Häufigkeiten zu den kleinsten noch vorkommenden gehören.
  - Man entfernt diese Paare aus M_i und fügt statt dessen das eine Paar $(X_1 \cup X_2, k_1 + k_2)$ hinzu. Das ergibt M_{i+1}.

Im Beispiel ergibt sich also

$$M_1 = \{\ (4,\{\texttt{a},\texttt{b}\})\ ,\ (3,\{\texttt{c}\})\ ,\ (3,\{\texttt{d}\})\ ,\ (5,\{\texttt{e}\})\ ,\ (7,\{\texttt{f}\})\ \}$$

- Als zweites fügt man dem schon konstruierten Teil des Graphen einen weiteren Knoten hinzu, markiert mit der Häufigkeit $k_1 + k_2$ und Kanten zu den Knoten, die für (X_1, k_1) und (X_2, k_2) eingefügt worden waren

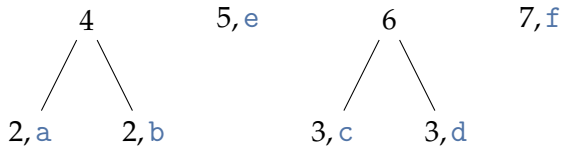
 © worsch&wacker 2008–2015



Da M_1 noch mehr als ein Element enthält, wiederholt man die beiden obigen Teilschritte:

$$M_2 = \{\ (4, \{\texttt{a}, \texttt{b}\})\ ,\ (6, \{\texttt{c}, \texttt{d}\})\ ,\ (5, \{\texttt{e}\})\ ,\ (7, \{\texttt{f}\})\ \}$$

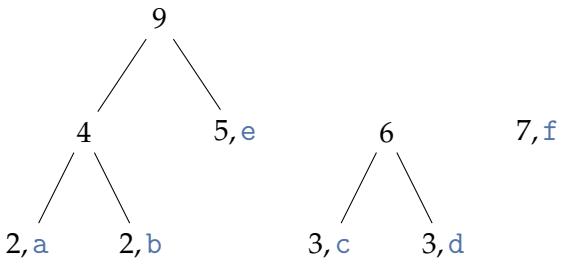
und der Graph sieht dann so aus:



Da M_2 noch mehr als ein Element enthält, wiederholt man die beiden obigen Teilschritte wieder:

$$M_3 = \{\ (9, \{\texttt{a}, \texttt{b}, \texttt{e}\})\ ,\ (6, \{\texttt{c}, \texttt{d}\})\ ,\ (7, \{\texttt{f}\})\ \}$$

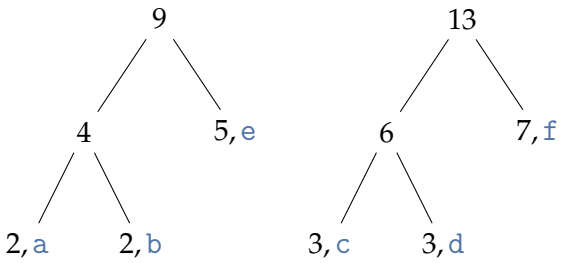
und der Graph sieht dann so aus:



Da M_3 noch mehr als ein Element enthält, wiederholt man die beiden obigen Teilschritte wieder:

$$M_4 = \{\ (9, \{\mathtt{a}, \mathtt{b}, \mathtt{e}\})\ ,\ (13, \{\mathtt{c}, \mathtt{d}, \mathtt{f}\})\ \}$$

und der Graph sieht dann so aus:

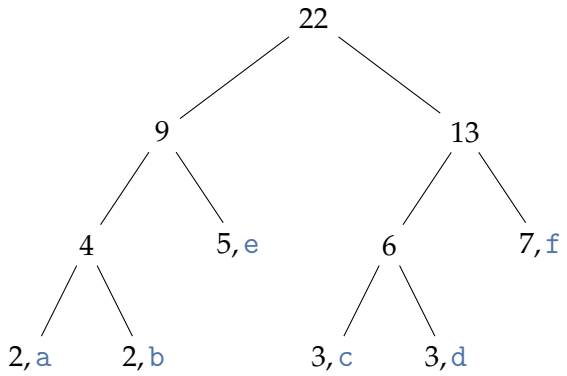


Zum Schluss berechnet man noch

$$M_5 = \{\ (22, \{\mathtt{a}, \mathtt{b}, \mathtt{c}, \mathtt{d}, \mathtt{e}, \mathtt{f}\})\ \}$$

und es ergibt sich der Baum

 © worsch&wacker 2008–2015



Aus diesem ergibt sich durch die Beschriftung der Kanten die Darstellung in Abbildung 8.2.

8.3.2 Weiteres zu Huffman-Codes

Wir haben das obige Beispiel so gewählt, dass immer eindeutig klar war, welche zwei Knoten zu einem neuen zusammengefügt werden mussten. Im allgemeinen ist das nicht so: Betrachten Sie einfach den Fall, dass viele Zeichen alle gleichhäufig vorkommen. Außerdem ist nicht festgelegt, welcher Knoten linker Nachfolger und welcher rechter Nachfolger eines inneren Knotens wird.

Konsequenz dieser Mehrdeutigkeiten ist, dass ein Huffman-Code nicht eindeutig ist. Das macht aber nichts: alle, die sich für ein Wort w ergeben können, sind „gleich gut“.

Dass Huffman-Codes gut sind, kann man so präzisieren: unter allen präfixfreien Codes führen Huffman-Codes zu kürzesten Codierungen. Einen Beweis findet man zum Beispiel unter `http://web.archive.org/web/20090917093112/http://www.maths.abdn.ac.uk/~igc/tch/mx4002/notes/node59.html`, 10.11.2015).

Zum Schluss wollen wir noch auf eine (von mehreren) Verallgemeinerung des oben dargestellten Verfahrens hinweisen. Manchmal ist es nützlich, nicht von den Häufigkeiten einzelner Symbole auszugehen und für die Symbole einzeln Codes zu berechnen, sondern für Teilwörter einer festen Länge $b > 1$. Alles wird ganz analog durchgeführt; der einzige Unterschied besteht darin, dass an den Blättern des Huffman-Baumes eben Wörter stehen und nicht einzelne Symbole.

Eine solche Verallgemeinerung wird bei mehreren gängigen Kompressionsverfahren (z. B. gzip, bzip2) benutzt, die, zumindest als einen Bestandteil, Huffman-Codierung benutzen.

Block-Codierung

Allgemein gesprochen handelt es sich bei diesem Vorgehen um eine sogenannte *Block-Codierung*. Statt wie bei einem Homomorphsimus die Übersetzung $h(x)$

jedes einzelnen Zeichens $x \in A$ festzulegen, tut man das für alle Teilwörter, die sogenannten Blöcke, einer bestimmten festen Länge $b \in \mathbb{N}_+$. Man geht also von einer Funktion $h : A^b \to B^*$ aus, und erweitert diese zu einer Funktion $h : (A^b)^* \to B^*$.

8.4 AUSBLICK

Wer Genaueres über UTF-8 und damit zusammenhängende Begriffe bei Unicode wissen will, dem sei die detaillierte Darstellung im *Unicode Technical Report UTR-17* empfohlen, den man unter http://www.unicode.org/reports/tr17/ (10.11.2015) im WWW findet.

Mehr über Bäume und andere Graphen werden wir in dem Kapitel mit dem überraschenden Titel „Graphen“ lernen.

 © worsch&wacker 2008–2015