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,
2. die die schöne Eigenschaft hat, dass jedes Ausgangswort und seine Übersetzung die gleiche Bedeutung haben.

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

10.1 VON WÖRTERN ZU ZAHLEN UND ZURÜCK

10.1.1 *Dezimaldarstellung von Zahlen*

Wir sind gewohnt, natürliche Zahlen im sogenannten Dezimalsystem notieren, das aus Indien kommt (siehe auch Einheit 5 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 Tabelle

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

  gegeben.
- 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

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

  ist, definiert man:

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

10.1.2 *Andere Zahldarstellungen*

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.
Schon Leibniz hat 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 (http://www.hs-augsburg.de/~harsch/germanica/Chronologie/17Jh/Leibniz/lei_bina.html, 9.10.2008) beschrieben und im Jahre 1703 in einer Zeitschrift veröffentlicht. Abbildung 10.1 zeigt Beispielrechnungen mit Zahlen in Binärdarstellung aus dieser Veröffentlichung.

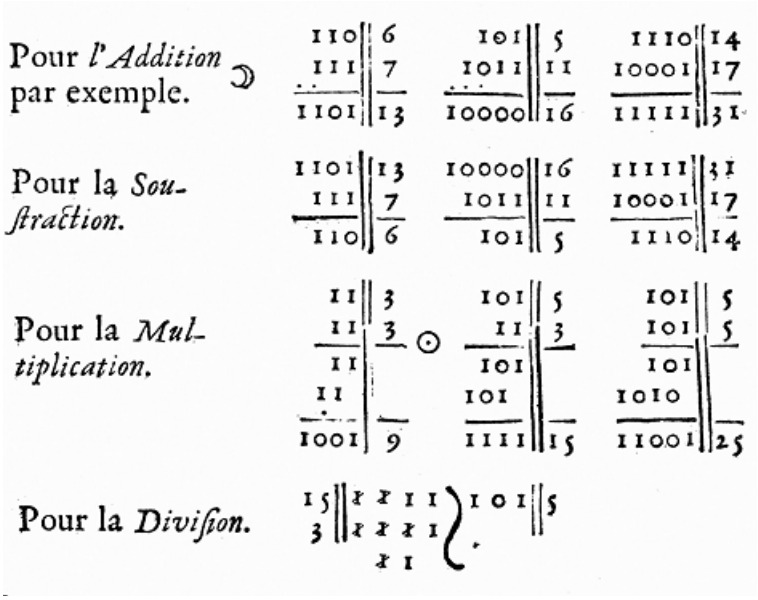


Abbildung 10.1: Ausschnitt aus dem Aufsatz „Explication de l'Arithmétique Binaire“ von Leibniz, Quelle: http://commons.wikimedia.org/wiki/Image:Leibniz_binary_system_1703.png (3.9.2008)

Binärdarstellung

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

$$\begin{aligned} \mathrm{num}_2(0) &= 0 \\ \mathrm{num}_2(1) &= 1 \\ \mathrm{Num}_2(\varepsilon) &= 0 \end{aligned}$$

sowie $$\forall w \in Z_2^* \; \forall x \in Z_2 : \mathrm{Num}_2(wx) = 2 \cdot \mathrm{Num}_2(w) + \mathrm{num}_2(x)$$

Damit ist dann z. B.

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

Hexadezimaldarstellung

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

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

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

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

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

10.1.3 *Von Zahlen zu ihren Darstellungen*

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.

Dazu benutzt man ein Alphabet Z_k mit k Ziffern, deren Bedeutungen die Zahlen in $\mathbb{G}_k$ sind. Für $i \in \mathbb{G}_k$ sei $\mathrm{repr}_k(i)$ dieses Zeichen. Die Abbildung repr_k ist also gerade die Umkehrfunktion zu num_k.

Gesucht ist dann eine Repräsentation von $n \in \mathbb{N}_0$ als Wort $w \in Z_k^*$ mit der Eigenschaft $\mathrm{Num}_k(w) = n$. 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 $\mathrm{Num}_k(w) = n$, dann auch $\mathrm{Num}_k(0w) = n$ (Beweis durch Induktion). Eine k-äre Darstellung einer Zahl kann man zum Beispiel so bestimmen:

$$
\begin{array}{l}
\text{// Eingabe: } n \in \mathbb{N}_0 \\
y \leftarrow n \\
w \leftarrow \varepsilon \\
m \leftarrow \begin{cases} 1 + \lfloor \log_k(n) \rfloor & \text{falls } n > 0 \\ 1 & \text{falls } n = 0 \end{cases} \\
\textbf{for } i \leftarrow 0 \textbf{ to } m - 1 \textbf{ do} \\
\quad r \leftarrow y \textbf{ mod } k \\
\quad w \leftarrow \mathrm{repr}_k(r) \cdot w \\
\quad y \leftarrow y \textbf{ div } k \\
\textbf{od} \\
\text{// am Ende: } n = \mathrm{Num}_k(w)
\end{array}
$$

Am deutlichsten wird die Arbeitsweise dieses Algorithmus, wenn man ihn einmal für einen vertrauten Fall benutzt: Nehmen wir $k = 10$ und $n = 4711$. Dann ist $m = 4$ und für jedes $i \in \mathbb{G}_5$ haben die Variablen r, w und y nach i Schleifendurchläufen die in der folgende Tabelle angegebenen Werte. Um die Darstellung besonders klar zu machen, haben wir die Fälle von $i = 0$ bis $i = 4$ von rechts nach links aufgeschrieben:

i	4	3	2	1	0
r	4	7	1	1	
w	4711	711	11	1	ε
y	0	4	47	471	4711

Die Schleifeninvariante $y \cdot 10^i + \mathrm{Num}_k(w) = n$ drängt sich förmlich auf. Und wenn man bewiesen hat, dass am Ende $y = 0$ ist, ist man fertig.

Der obige Algorithmus liefert das (es ist eindeutig) kürzeste Wort $w \in Z_k^*$ mit $\mathrm{Num}_k(w) = n$. Dieses Wort wollen wir auch mit $\mathrm{Repr}_k(n)$ bezeichnen. 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.

10.2.1 *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` 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.

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

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

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. Eine grundlegende Einführung in solche Fragestellungen können Sie in Vorlesungen über die Semantik von Programmiersprachen bekommen.

Warum macht man Übersetzungen? Zumindest die folgenden vier 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).
- *Kompression:* Manchmal führen Übersetzungen zu kürzeren Texten, die also weniger Platz benötigen. Und zwar *ohne* zu einem größeren Alphabet überzugehen. Wir werden im Abschnitt 10.3 über Huffman-Codes sehen, warum und wie das manchmal möglich ist.
- *Verschlüsselung:* Im Gegensatz zu verbesserter Lesbarkeit übersetzt man manchmal gerade deshalb, um die Lesbarkeit möglichst unmöglich zu machen, jedenfalls für Außenstehende. Wie man das macht, ist Gegenstand von Vorlesungen über Kryptographie.
- *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 $\mathrm{sem}_A(w) = \mathrm{sem}_B(f(w))$ 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. M. a. W., f ist injektiv. In diesem Fall kann man dann die Bedeutung sem_B im wesentlichen *definieren* durch die Festlegung $\mathrm{sem}_B(f(x)) = \mathrm{sem}_A(x)$. 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 $\mathrm{sem}_A(x) \neq \mathrm{sem}_B(x)$ aber die Funktionswerte $f(x) = f(y) = z$, dann wäre nicht klar, was $\mathrm{sem}_B(z)$ sein soll.

Codierung

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

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 gleich bzw. am Ende dieser Einheit eingehen werden.

10.2.2 *Homomorphismen*

Es seien A und B zwei Alphabete und $h : A \to B^*$ eine Abbildung. Zu h kann man in der Ihnen inzwischen vertrauten Art eine Funktion $h^{**} : A^* \to B^*$ definieren vermöge

$$h^{**}(\varepsilon) = \varepsilon$$
$$\forall w \in A^* : \forall x \in A : h^{**}(wx) = h^{**}(w)h(x)$$

Eine solche Abbildung h^{**} nennt man einen *Homomorphismus*. Häufig erlaubt man sich, statt h^{**} einfach wieder h zu schreiben (weil für alle $x \in A$ gilt: $h^{**}(x) = h(x)$).

h^{**}

Homomorphismus

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

ε-freier Homomorphismus

Dass eine Abbildung $h : A \to B^*$ eine Codierung, also eine injektive Abbildung $h : A^* \to B^*$ induziert, ist im allgemeinen nicht ganz einfach zu sehen. Es gibt aber einen Spezialfall, in dem 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 $h(x_2)$.

präfixfrei

Das praktische Beispiel schlechthin für einen Homomorphismus ist die Repräsentation des ASCII-Zeichensatzes (siehe Unterabschnitt 3.1.1) im Rechner. Das geht einfach so: Ein Zeichen x mit der Nummer n im ASCII-Code wird codiert durch dasjenige Byte $w \in \{0,1\}^8$, für das $\mathrm{Num}_2(w) = n$ ist. Und längere Texte werden übersetzt, indem man nacheinander jedes Zeichen einzeln so abbildet.

Dieser Homomorphismus hat 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, wo das nicht so ist.

10.2.3 *Beispiel Unicode: UTF-8*

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. Das es so viele Zeichen sind (und 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 dem RFC 3629 (`http://www.rfc-editor.org/rfc/rfc3629.txt` (1.10.08))

> 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	0xxxxxxx
0000 0080 - 0000 07FF	110xxxxx 10xxxxxx
0000 0800 - 0000 FFFF	1110xxxx 10xxxxxx 10xxxxxx
0001 0000 - 0010 FFFF	11110xxx 10xxxxxx 10xxxxxx 10xxxxxx

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 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.

10.3 HUFFMAN-CODIERUNG

Es sei ein Alphabet A und ein Wort $w \in A^*$ gegeben. Eine sogenannte *Huffman-Codierung* von w ist eine Abbildung $h : A^* \to Z_2^*$, die ein ε-freier Homomorphismus ist. Es ist also h eindeutig festgelegt durch die Funktionswerte $h(x)$ für alle $x \in A$. *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.

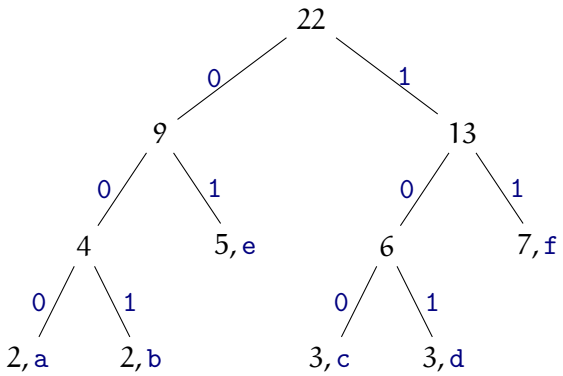


Abbildung 10.2: Ein Beispielbaum für die Berechnung eines Huffman-Codes.

10.3.1 *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 folgende Baum ergeben:

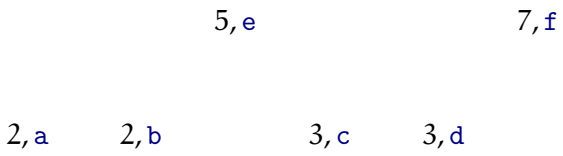
In diesem Fall ist dann der Homomorphismus gegeben durch

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:

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

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



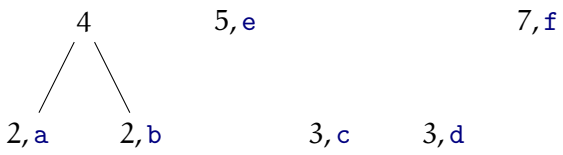
Solange eine Menge M_i noch mindestens zwei Paare enthält, tut man folgendes:

- 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

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

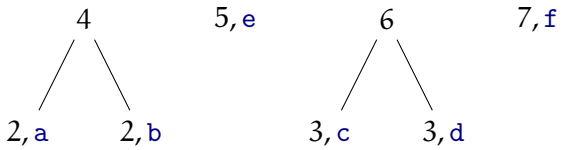
- Als zweites fügt man dem schon konstruierten Teil des Graphen einen weiteren Knoten hinzu, markiert mit der Häufigkeit $k_1 + k_2$ und Kanten zu den Knoten, die für (X_1, k_1) und (X_2, k_2) eingefügt worden waren



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

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

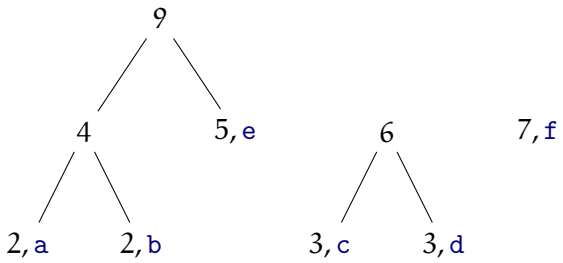
und der Graph sieht dann so aus:



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

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

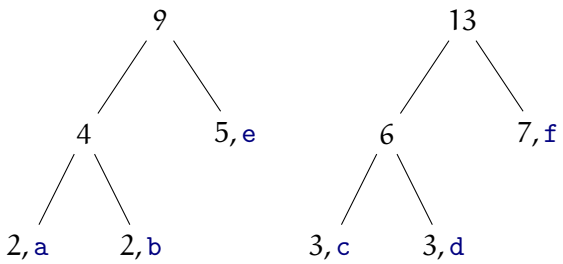
und der Graph sieht dann so aus:



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

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

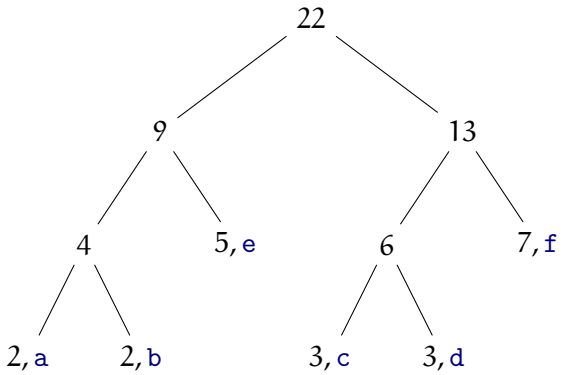
und der Graph sieht dann so aus:



Zum Schluss berechnet man noch

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

und es ergibt sich der Baum



Aus diesem ergibt sich durch die Beschriftung der Kanten die Darstellung in Abbildung 10.2.

10.3.2 *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://www.maths.abdn.ac.uk/~igc/tch/mx4002/notes/node59.html` (24.11.08).

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.

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

Block-Codierung

also von einer Funktion $h : A^b \to B^*$ aus, und erweitert diese zu einer Funktion $h : (A^b)^* \to B^*$.

10.4 AUSBLICK

Wir haben uns in dieser Einheit nur mit der Darstellung von nichtnegativen ganzen Zahlen beschäftigt. Wie man bei negativen und gebrochenen Zahlen z. B. in einem Prozessor vorgeht, wird in einer Vorlesung über technische Informatik behandelt werden.

Wer Genaueres über UTF-8 und damit zusammenhängende Begriffe bei Unicode wissen will, sei die detaillierte Darstellung im *Unicode Technical Report UTR-17* empfohlen, den man unter http://www.unicode.org/reports/tr17/ (9.10.2008) im WWW findet.

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