Dr. David Sabel
Institut für Informatik

Fachbereich Informatik und Mathematik
Johann Wolfgang Goethe-Universität Frankfurt am Main

Praktikum BKSPP

Sommersemester 2012

Aufgabenblatt Nr. 2

Abgabe: Mittwoch, 16. Mai 2012

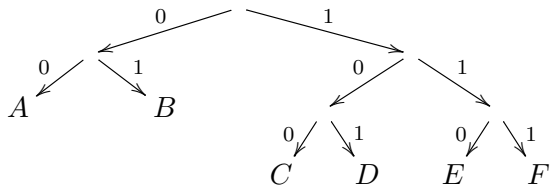
1 Zeichenbasierte Textkompression

Eine Zeichenbasierte Kodierung ist möglich mithilfe von Codebäumen: Sei $\mathcal{A}$ ein Alphabet, dann ist ein *Codebaum* zu $\mathcal{A}$ ein binärer Baum, so dass gilt:

- Jedes Blatt ist mit einem Symbol aus $\mathcal{A}$ markiert, wobei jedes Symbol aus $\mathcal{A}$ genau an einem Blatt als Markierung vorkommt.
- Ein innerer Knoten hat stets zwei Kinder, wobei eine Kante zu einem Kind mit 0 und die andere Kante zum anderen Kind mit 1 markiert ist.

Das *Codewort* $c(a)$ eines Symbols $a \in \mathcal{A}$ erhält man, indem man die Kantenmarkierungen des (eindeutigen) Pfades von der Wurzel zum mit a markierten Blatt konkateniert. Offensichtlich ist die Kodierung mit einem Codebaum eindeutig und Präfix-frei, d.h. für je zwei Symbole $a, a' \in \mathcal{A}$ gilt weder, dass $c(a)$ ein Präfix von $c(a')$ noch, dass $c(a')$ ein Präfix von $c(a)$ ist.

Wir betrachten als Beispiel den folgenden Codebaum für das Alphabet $\{A, B, C, D, E, F\}$



Die Codeworte, die sich daraus für die einzelnen Buchstaben ergeben, sind in der folgenden Tabelle eingetragen:

a	$c(a)$
A	00
B	01
C	100
D	101
E	110
F	111

Kodiert man den Text "ABCDEFFA" anhand des obigen Codebaumes, so erhält man als Code 000110010111011111100.

Aufgabe 1 *Implementieren Sie ein Modul* `Codebaum`*, welches einen (rekursiven) Datentyp* `Codebaum a` *für Codebäume zur Verfügung stellt und polymorph über einem Alphabet definiert ist. Im „Hintergrundwissen" wird bereits ein Datentyp* `Baum` *definiert, an dem Sie sich orientieren können, wobei Markierungen der inneren Knoten nicht notwendig sind.*

Programmieren Sie eine Funktion `mkGetCode :: Codebaum a -> a -> [Int]`*, die aus einem Codebaum eine* Funktion *erstellt, die es erlaubt $c(a)$ zu berechnen (hierbei ist der Code eine Liste bestehend aus Nullen und Einsen). Die durch* `mkGetCode` *erstellte Funktion sollte möglichst effizient implementiert werden, d.h. zumindest pro Zugriff höchstens lineare Zeit in der Größe des Alphabets benötigen.*

Programmieren Sie eine Funktion `kodiere :: (Codebaum a) -> [a] -> [Int]`*, die einen Codebaum und ein Wort erhält und den Code des Wortes entsprechend des Codebaumes berechnet. Verwenden Sie die Funktion* `mkGetCode` *für diese Implementierung.*

Wie lautet der Code für das Wort

```
ABCDEFFFFFFFEDCBAAAAAAAAAAAAAABCCCDDDAAAABBBBCDCCAAACCCC
```

wenn Sie obigen Codebaum verwenden?

1.1 Die Huffman-Kodierung

Eine *optimale* Zeichenbasierte Kompression liefert die Huffman-Kodierung.

Die Huffman-Kodierung versucht nun einen Codebaum zu erzeugen, der optimal ist, d.h. der für einen gegebenen Text den kürzesten Code liefert (wobei die einzelnen Zeichen durch ihr Codewort repräsentiert werden). Dabei wird versucht möglichst kurze Codes für häufig vorkommende Zeichen zu verwenden, während selten vorkommende Zeichen im Gegenzug durch längere Bitfolgen kodiert werden.

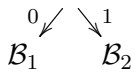
Sei $\mathcal{T}$ ein Text (d.h. eine Folge von Symbolen über einem Alphabet $\mathcal{A}$). Dann geht die Huffman-Kodierung wie folgt vor:

1. Berechne die *relative Häufigkeit* r der Symbole $a \in \mathcal{A}$ im Text $\mathcal{T}$, d.h. welchen Anteil das Symbol a am Text T hat, oder als Formel ausgedrückt:

$$r(a, \mathcal{T}) = \frac{\text{Anzahl der Vorkommen von } a \text{ in } \mathcal{T}}{\text{Anzahl der Zeichen von } \mathcal{T}}$$

2. Erzeuge einen *Wald* von Codebäumen: Für jedes Symbol $a \in \mathcal{A}$ erzeuge einen Codebaum bestehend aus einem Blatt markiert mit a. Markiere jeden Baum mit einer relativen Häufigkeit. Diese ist initial die relative Häufigkeit des entsprechenden Symbols am Blatt.

3. Iteriere den folgenden Schritt solange, bis ein einziger Codebaum entstanden ist: Wähle aus dem Wald der Codebäume die beiden Bäume aus, die mit der *kleinsten* relativen Häufigkeit markiert sind. Seien $\mathcal{B}_1$, $\mathcal{B}_2$ diese Bäume. Verschmelze die beiden Bäume zu einem einzigen Baum folgendermaßen:



   Wenn $\mathcal{B}_1$ und $\mathcal{B}_2$ mit den relativen Häufigkeiten R_1 und R_2 markiert sind, so ist der neue Baum mit der Häufigkeit $R_1 + R_2$ markiert.

Wir betrachten als Beispiel den String $\mathcal{T}$ ="`AECACEFAABDDBEAEAEECEEEDF`". Die relativen Häufigkeiten der einzelnen Zeichen lassen sich berechnen als:

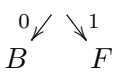
a	Anzahl a im Text $\mathcal{T}$	$r(a, \mathcal{T})$
A	6	$6/25 = 0,24 = 24\%$
B	2	$2/25 = 0,08 = 8\%$
C	3	$3/25 = 0,12 = 12\%$
D	3	$3/25 = 0,12 = 12\%$
E	9	$9/25 = 0,36 = 36\%$
F	2	$2/25 = 0,08 = 8\%$

Nun wird zunächst der Wald erzeugt, der jeweils einen Baum mit einem Blatt markiert mit einem Zeichen enthält, zusätzlich wird für jeden Baum die relative Häufigkeit gespeichert. Wir stellen dies durch Paare dar.

$$W_0 = \{(A, 24\%), (B, 8\%), (C, 12\%), (D, 12\%), (E, 36\%), (F, 8\%)\}$$

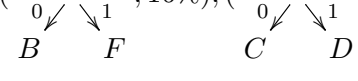
In der ersten Iteration werden die Bäume für B und F vereint. Der neue Wald ist:

$$W_1 = \{(A, 24\%), (\ \ , 16\%), (C, 12\%), (D, 12\%), (E, 36\%)\}$$



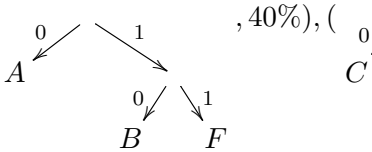
Nun werden die Bäume für C und D vereint:

$$W_2 = \{(A, 24\%), (\ \ , 16\%), (\ \ , 24\%), (E, 36\%)\}$$

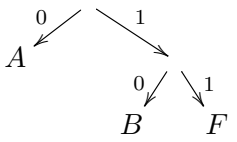
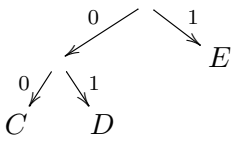


Im nächsten Schritt muss der mit 16% markierte Baum mit einem der beiden mit 24% markierten Bäume vereint werden, wir wählen den ersten:

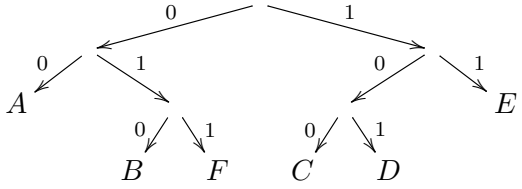
$$W_3 = \{(\ \ , 40\%), (\ \ , 24\%), (E, 36\%)\}$$



Der nächste Schritt ergibt:

$W_4 = \{($  $, 40\%), ($  $, 60)\}$

Die letzte Vereinigung ergibt den Codebaum:



Damit lässt sich der Text $\mathcal{T}$ als

```
00111000010011011000001010110101011001100111110011111110101 1
```

kodieren. Wenn man je 8 dieser Bits als ASCII-Code auffasst, ergibt dies einen String der Länge 8. Da der ursprüngliche String die Länge 25 hatte, wurde die Eingabe auf 32% seiner Größe komprimiert.

Aufgabe 2 *Implementieren Sie ein Modul* `Huffman` *mit einer Funktion* `text2Huffman :: String -> (Codebaum Char,[Int])`*, welche einen Text entsprechend der Huffmankodierung in eine Bitfolge kodiert und als Ergebnis das Paar bestehend aus Codebaum und Code liefert. Implementieren Sie anschließend ein Konsolen-Programm, dass einen Text entsprechend der Huffmankodierung kodieren kann und in einer Datei abspeichern kann, sowie so kodierte Dateien lesen und dekodieren kann.*

Hinweise:

- *Beim Abspeichern des Codes sollten Sie nicht die 0-1-Folge in die Datei schreiben, sondern diese wieder in ASCII-Buchstaben umwandeln, um wirklich eine Kompression zu erzielen. Überlegen Sie sich ein geeignetes Vorgehen für diese Konvertierung und implementieren Sie diese. Ein mögliches Vorgehen ist es je 8 Bit als ein Zeichen vom Typ* `Char` *zu interpretieren.*
- *Beim Abspeichern müssen Sie für ein späteres Dekodieren den Huffman-Baum ebenfalls in einem sinnvollen Format abspeichern.*

2 Stringersatzverfahren als Kompressionsalgorithmen

Während die Huffman-Kodierung eine Entropiekodierung auf den Zeichen ist, gibt es so genannte *Stringersatzverfahren*, die Wiederholungen von Folgen des Eingabestrings erkennen und diese zusammenfassen, und dadurch eine Kompression erzielen.

2.1 Kompression durch Run Length Encoding

Ein einfaches Verfahren ist die so genannte „Run Length Encoding“. Bei diesem Verfahren werden Teilfolgen gleicher Zeichen durch einen Zähler und das Zeichen ersetzt. Z.B. wird der Text `"aaaaabaabaaaccccccddaaaa"` durch `"5ab2ab3a5c2d4a"` dargestellt (einzelne Zeichen werden nicht durch Zähler versehen). Damit die Dekompression möglich ist, dürfen hierbei entweder Ziffern nicht in der Eingabe vorkommen oder in der Ausgabe müssen Trennsymbole eingefügt werden (diese dürfen in der Eingabe nicht vorkommen), z.B. `"#5:ab#2:ab#3:a#5:c#2:d#4:a"`.

Aufgabe 3 *Implementieren Sie in Haskell ein Modul* `RLE`*, welches Funktionen zur Kodierung und Dekodierung von Strings mittels des „Run Length Encoding“-Verfahrens zur Verfügung stellt. Treffen Sie angemessene Maßnahmen, um eindeutiges Dekodieren zu ermöglichen. Verknüpfen Sie das RLE-Verfahren mit dem Huffman-Verfahren in geeigneter Weise.*

2.2 Kompression durch das LZ77-Verfahren

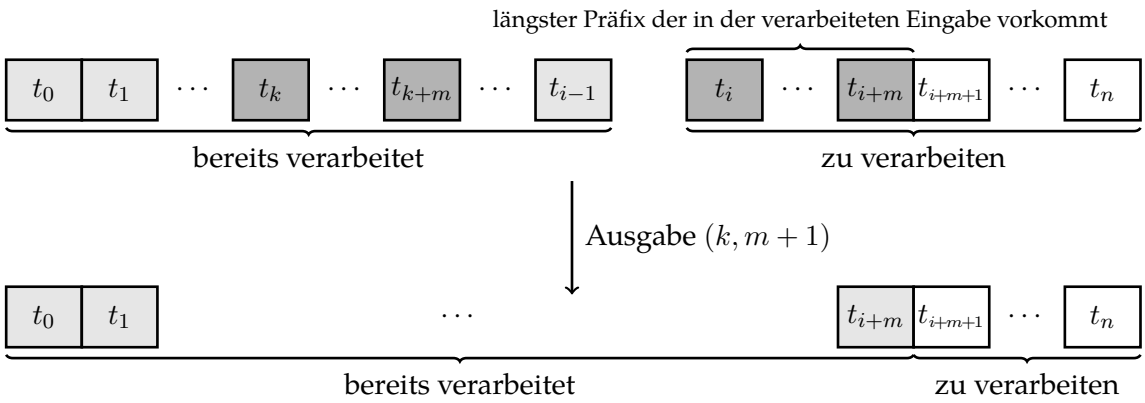
Ein i.A. wesentlich besseres Verfahren bietet die so genannte Lempel-Ziv-Komprimierung[1]. Diese versucht bereits gelesene Teilfolgen der Eingabe bei wiederholtem Auftreten durch eine Referenz auf die ursprüngliche Teilfolge zu ersetzen.

Sei $t_0 \dots t_n$ der Eingabestring, wobei die Eingabe $t_0 \dots t_{i-1}$ schon verarbeitet sei, d.h. der Algorithmus muss noch den String $t_i \dots t_n$ komprimieren. Der Algorithmus geht nun wie folgt vor:

1. Sei $t_i \dots t_{i+m}$ der *längste Präfix* von $t_i \dots t_n$, der bereits als Substring in $t_0 \dots t_{i-1}$ vorkommt. Nehmen wir an, dies sei der String $t_k \dots t_{k+m}$ (wobei $t_p = t_q$ für alle $(p, q) \in \{(i, k), (i + 1, k + 1), \dots, (i + m, k + m)\}$).
2. Wenn der Präfix leer ist (d.h. es gibt keinen echten Präfix der schon als Teilstring) in der verarbeiteten Folge vorkommt, dann speichert der Algorithmus das Symbol t_i ab und fährt mit der Resteingabe $t_{i+1} \dots t_n$ mit Schritt 1 fort.
3. Anderenfalls erzeugt der Agorithmus anstelle des Präfixes $t_i \dots t_{i+m}$ das Zahlenpaar $(k, m+1)$, d.h. er merkt sich statt des Teilstrings $t_i \dots t_{i+m}$ die Referenz auf die schon bekannte Eingabe.
4. Danach fährt der Algorithmus mit der Resteingabe $t_{i+m+1} \dots t_n$ mit Schritt 1 fort.

[1] genauer: Wir betrachten eine Variante der LZ77-Kodierung

Der Schritt des nicht-leeren Präfixes kann illustriert werden durch:



Wir betrachten als Beispiel den Text `"aabaaabaababaaaababbbaabbbbccaaa"`:

Verarbeitete Eingabe	**Resteingabe**	**Ausgabe**	**Erkannter Präfix**
	aabaaabaababaaaababbbaabbbbccaaa	a	ε
a	abaaabaababaaaababbbaabbbbccaaa	(0,1)	a
aa	baaabaababaaaababbbaabbbbccaaa	b	ε
aab	aaabaababaaaababbbaabbbbccaaa	(0,2)	aa
aabaa	abaababaaaababbbaabbbbccaaa	(1,4)	abaa
aabaaabaa	baaabaababbbaabbbbccaaa	(2,2)	ba
aabaaabaaba	baaababbbaabbbbccaaa	(2,6)	baaaba
aabaaabaababaaaba	bbbaabbbbccaaa	(2,1)	b
aabaaabaababaaaab	bbaabbbbccaaa	(2,1)	b
aabaaabaababaaabb	baabbbbccaaa	(6,4)	baab
aabaaabaababaaababbbaab	bbbccaaa	(17,3)	bbb
aabaaabaababaaababbbaabbbb	ccaaa	c	ε
aabaaabaababaaababbbaabbbbc	caaa	(26,1)	c
aabaaabaababaaababbbaabbbbcc	aaa	(3,3)	aaa
aabaaabaababaaababbbaabbbbccaaa			

Der komprimierte Text lautet daher

`a(0,1)b(0,2)(1,4)(2,2)(2,6)(2,1)(2,1)(6,4)(17,3)c(26,1)(3,3).`

Teilstrings der Länge 1 wurden hierbei auch durch Stringreferenzen dargestellt, was sich jedoch im Grunde nicht lohnt, da das Zeichen vermutlich weniger Platz als die Angabe der beiden Zahlen benötigt. D.h. ein bessere Kodierung wäre vermutlich `aab(0,2)(1,4)(2,2)(2,6)bb(6,4)(17,3)cc(3,3)`. Die Dekomprimierung baut den Text von links nach rechts wieder auf, wobei die bereits erstellte Ausgabe als Suchraum für die noch zu erstellende Ausgabe dient:

Komprimierter Reststring	**Ausgabe**
aab(0,2)(1,4)(2,2)(2,6)bb(6,4)(17,3)cc(3,3)	ε
ab(0,2)(1,4)(2,2)(2,6)bb(6,4)(17,3)cc(3,3)	a
b(0,2)(1,4)(2,2)(2,6)bb(6,4)(17,3)cc(3,3)	aa
(0,2)(1,4)(2,2)(2,6)bb(6,4)(17,3)cc(3,3)	aab
(1,4)(2,2)(2,6)bb(6,4)(17,3)cc(3,3)	aabaa
(2,2)(2,6)bb(6,4)(17,3)cc(3,3)	aabaaabaa
(2,6)bb(6,4)(17,3)cc(3,3)	aabaaababaa
bb(6,4)(17,3)cc(3,3)	aabaaabaababaaaba
b(6,4)(17,3)cc(3,3)	aabaaabaababaaabab
(6,4)(17,3)cc(3,3)	aabaaabaabababaaabab
(17,3)cc(3,3)	aabaaabaababaaabababbbaab
cc(3,3)	aabaaabaababaaabababbbaabbbb
c(3,3)	aabaaabaababaaabababbbaabbbbc
(3,3)	aabaaabaababaaabababbbaabbbbcc
ε	aabaaabaababaaabababbbaabbbbccaaa

Aufgabe 4 *Implementieren Sie in Haskell ein Modul LZ, welches Funktionen* `lz77Compress` *und* `lz77Decompress` *implementiert, die die LZ77 Kompression und Dekompression durchführen. Entwerfen Sie ein Konsolenprogramm, welches ermöglicht, Dateien mit LZ77 zu komprimieren und zu dekomprimieren.*

Verknüpfen Sie die Huffman-Kodierung mit der LZ77-Kompression indem Sie beide Kompressionsverfahren in geeigneter Weise miteinander kombinieren.

Optimieren Sie die LZ77-Kodierung derart, dass nur dann Stringreferenzen angelegt werden, wenn die Stringreferenz (das Paar der beiden Zahlen) weniger Platz einnimmt als der ursprüngliche Eingabestring.

2.3 Kompression durch das LZW-Verfahren

Ein ähnlicher Kompressionsalgorithmus ist der LZW-Algorithmus (Lempel-Ziv-Welch). Hierbei baut der Algorithmus sukzessive ein Wörterbuch schon bekannter Strings auf und benutzt das Wörterbuch zum Einen zum Komprimieren der Resteingabe, zum Anderen zum Hinzufügen neuer Wörter in das Wörterbuch. Sei D das Wörterbuch und $t_i \dots t_n$ die Resteingabe. Der LZW-Algorithmus berechnet zunächst den längsten Präfix von $t_i \dots t_n$, der bereits im Wörterbuch vorhanden ist. Sei dies der String $t_i \dots t_m$ mit dem Schlüssel k in D. Der Präfix $t_i \dots t_m$ wird nun durch k ersetzt und falls $m < n$ wird der String $t_i \dots t_{m+1}$ neu ins Wörterbuch eingetragen. Damit das Wörterbuch kompakt bleibt, genügt es hierbei, sich im Wörterbuch lediglich k, t_{m+1} und den neuen Schlüssel zu merken. Wir nehmen des Weiteren an, dass sämtliche Zeichen der Eingabe schon im anfänglichen Wörterbuch abgespeichert sind.

Wir betrachten erneut den String `aabaaabaababaaababbbaabbbbccaaa` und nehmen an, dass `a`, `b`, und `c` bereits in das Wörterbuch eingefügt wurden (mit den Schlüsseln 1,2,3).

Wörterbuch					
Schlüssel	**Präfix**	**Suffix**	Wort		
1		a	a		
2		b	b		
3		c	c		
				Resteingabe	**Ausgabe**
				aabaaabaababaaabababbbaabbbbccaaa	1
4	1	a	aa	abaaabaababaaabababbbaabbbbccaaa	1
5	1	b	ab	baaabaababaaabababbbaabbbbccaaa	2
6	2	a	ba	aaabaababaaabababbbaabbbbccaaa	4
7	4	a	aaa	abaababaaabababbbaabbbbccaaa	5
8	5	a	aba	aababaaabababbbaabbbbccaaa	4
9	4	b	aab	babaaabababbbaabbbbccaaa	6
10	6	b	bab	baaababbbaabbbbccaaa	6
11	6	a	baa	aababbbaabbbbccaaa	9
12	9	a	aaba	abbbaabbbbccaaa	5
13	5	b	abb	bbaabbbbccaaa	2
14	2	b	bb	baabbbbccaaa	11
15	11	b	baab	bbbbccaaa	14
16	14	b	bbb	bbccaaa	14
17	14	c	bbc	ccaaa	3
18	3	c	cc	caaa	3
19	3	a	ca	aaa	7

Der String wurde daher in den Code `1,1,2,4,5,6,6,9,5,2,11,14,14,3,3,7` kodiert, wobei das Wörterbuch lautet:

$$\{1 \mapsto a, 2 \mapsto b, 3 \mapsto c, 4 \mapsto 1a, 5 \mapsto 1b, 6 \mapsto 2a, 7 \mapsto 4a, 8 \mapsto 5a, 9 \mapsto 4b, 10 \mapsto 6b, 11 \mapsto 6a,$$
$$12 \mapsto 9a, 13 \mapsto 5b, 14 \mapsto 2b, 15 \mapsto 11b, 16 \mapsto 14b, 17 \mapsto 14c, 18 \mapsto 3c, 19 \mapsto 3c\}$$

Hierbei kann man das Wörterbuch auch als kontextfreie Grammatik auffassen und die Ausgabe als Folge von Nichtterminalen (als rechte Seite einer Regel für das Startsymbol). Das Dekomprimieren entspricht dann gerade der Herleitung eines Wortes. Wie das Beispiel verdeutlicht, ist es jedoch nicht nötig die gesamte Ausgabefolge zu speichern, da diese (bis auf die letzte Zahl (7)) bereits in der kontextfreien Grammatik (die Präfix-Einträge) ablesbar sind.

Aufgabe 5 *Implementieren Sie den LZW-Algorithmus in Haskell. Während der Kompression muss im Wörterbuch quasi „falsch herum" gesucht werden, denn es soll der Schlüssel eines möglichst langen Präfixes der Resteingabe gesucht werden. Benutzen Sie für diese Suche eine sinnvolle und effiziente Datenstruktur. Verknüpfen Sie den LZW-Algorithmus in geeigneter Weise mit der Huffman-Kodierung. Implementieren Sie ein Konsolenprogramm für die LZW-Kompression und Dekompression.*