Beleg 1 - Aufgabenblatt

Vorbereitung:

Laden Sic sich das vorbereitete Erlang-Dateien (bel1 und bel1_tests) herunter. Kompilieren Sie die Dateien. Tests werden mit dem Kommando eunit:test(bel1) ausgeführt. Ergänzen Sie die vorhandenen Funktionsrümpfe.

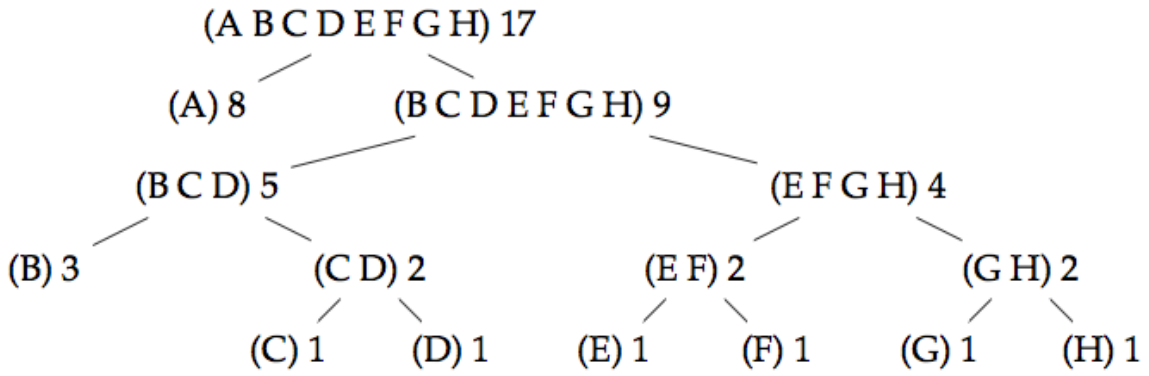
Implementierung des Huffman-Coding-Verfahrens

Huffman-Kodierung ist ein Verfahren für die Kompression von Texten.

In einem normalen, nicht komprimierten Text werden Zeichen durch eine feste Anzahl von Bits repräsentiert. So benötigt bspw. ein Zeichen in ASC II-Codierung genau 8 Bits. Da nur ein Teil der Zeichen in der ASC II-Codierung wirklich darstellbare Characters sind, kann mit einer Umkodierung eine wesentliche Reduktion der Datenmenge erreicht werden.

Bei der Huffman-Kodierung kann jedes Character eine Codierung mit variabler Länge besitzen. Die Länge richtet sich dabei nach der Anzahl der Vorkommen innerhalb eines Textes: Zeichen die häufig auftreten haben eine niedrigere Länge als Zeichen die seltener vorkommen. Dabei definiert der Huffman-Code die speziellen Bit-Sequenzen, die für jedes Zeichen verwendet werden.

Ein Huffman-Code kann durch einen Binärbaum repräsentiert werden: Die Blätter entsprechen den Zeichen, die kodiert werden sollen, während der Weg zu den Blättern über die Knoten die Kodierungssequenz darstellt. Der hier dargestellte Binärbaum entspricht der Kodierung der Buchstaben A bis H mit Angabe der korrespondierenden Häufigkeiten.



Achtung: Der hier aufgemalte Baum dient nur dem Verständnis – der in den Tests aufgeführte Baum wird nach anderen Kriterien aufgebaut.

Jedes Blatt besitzt ein Gewicht, das die Häufigkeit des Vorkommens darstellt. In dem Beispiel hat A das höchste Gewicht, da der Buchstabe 8 mal in einem Text vorgekommen ist. Das geringste Gewicht haben die Buchstaben C, D, E, F, G, H mit einer Häufigkeit von 1.

Jeder innerer Knoten im Baum repräsentiert die Zeichen der darunter liegenden Blätter. Das Gewicht ist dabei die Summe aller Buchstabenvorkommen der darunter liegenden Blätter. Diese Information ist erforderlich, um einen optimalen Huffman-Code aufzubauen, d.h. den

Binärbaum zu konstruieren (Ein Baum ist nur dann optimal, wenn die Häufigkeit der Vorkommen mit den Häufigkeiten im zu kodierenden Text übereinstimmt).

Der Baum hat eine rekursive Struktur. Das bedeutet, dass jeder Teilbaum ein valider Code für ein kleineres Alphabets ist.

Kodierung:

Die Kodierung findet über die Traversierung des Baums statt. Jede Verzweigung nach links bedeutet eine 0 - jede Verzweigung nach rechts bedeutet eine 1. Das heißt, bei der Kodierung für einen gegebenen Huffman-Baum wird für jedes Zeichen der Baum einmal durchlaufen und für jeden Branch nach links eine 0 eingefügt und für jeden Branch nach rechts eine 1. In dem Beispiel würde aus der Zeichenkette aaabac 00010001010 kodiert werden.

Dekodierung

Die Dekodierung dreht das Verfahren um. Der Baum wird so lange von der Wurzel durchlaufen, bis ein Blatt erreicht wurde. Wurde ein Blatt erreicht, so kann der mit dem Blatt assoziierte Buchstabe in das Ergebnis übernommen werden. Danach wird der Baum wieder von oben durchlaufen. Ein Beispiel wäre die Sequenz 10001010, die zu BAC dekodiert werden würde.

Implementierung

Ein Huffman-Baum kann in Erlang folgendermaßen repräsentiert werden:

```
-type tree():: fork() | leaf().
-type fork() :: #fork{}.
-type leaf() :: #leaf{}.
-type bit() :: 0 | 1.

-record(fork, {left::tree(), right::tree(), chars::list(char()),
           weight::non_neg_integer()}).

-record(leaf, {char::char(), weight::non_neg_integer()}).

-spec weight(tree()) -> non_neg_integer().
weight(#fork{weight=W}) -> W;
weight(#leaf{weight=W}) -> W.

-spec chars(tree()) -> list(char()).
chars(#fork{chars=C}) -> C;
chars(#leaf{char=C}) -> [C].

-spec makeCodeTree( T1::tree(), T2::tree()) -> tree().
makeCodeTree(T1 , T2) -> case (chars(T1) < chars(T2)) of
      true -> #fork{left=T1, right=T2, chars=chars(T1)++chars(T2),
                  weight=weight(T1)+weight(T2)};
      false -> #fork{left=T2, right=T1, chars=chars(T2)++chars(T1),
                  weight=weight(T1)+weight(T2)}
end.
```

Für den Code-Baum wurden zwei Records namens fork (innerer Knoten) und leaf (Blatt) definiert. Weiter wurden eine Funktion weight für die Rückgabe des Gewichts eines Blatts bzw. inneren Knoten definiert sowie eine Funktion chars, die eine Liste von Character

zurückgibt, die durch den Knoten repräsentiert wird. MakeCodeTree generiert über die Teilbäume einen Baum, in dem die Liste der Character zusammengeführt wird sowie das Gewicht zusammengezählt. Hier erfolgt eine Fallunterscheidung bei der Anordnung des linken und rechten Teilbaums (zur Verbesserung der Testbarkeit). Der Knoten dessen Buchstaben lexikalisch vor dem anderen liegt, wird auf die linke Seite gelegt.

Die Definition der Typen ist nicht zwingend erforderlich. Sie ermöglicht eine statische Typprüfung mit dem Dialyzer-Tool durchzuführen. Die im Text beschriebenen Typen entsprechen keinem gültigen Erlang-Code - dieser ist in den Dateien zu finden. Weiter sind zahlreiche Unit-Tests definiert, die mit dem Kommando eunit:test(bel1) aufgerufen werden.

Ein Baum kann über die Funktion makeCodeTree folgendermaßen erzeugt werden:

```
SampleTree = makeCodeTree(#leaf{char=$x, weight=1}),#leaf{char=$x,
weight=1})
```

Konstruktion eines Huffman-Baums

Der erste Schritt im Huffman-Verfahren ist die Ermittlung des optimalen Huffman Codes. Dabei ist wichtig, dass das Verfahren verlustfrei abläuft, d.h. die Rekonstruktion des Originaltexts zum selben Ergebnis kommt.

`Implementieren` Sie eine Funktion `createCodeTree` mit der folgenden Signatur:
`createCodeTree(Text)`

Verwenden Sie dazu die folgende Vorgehensweise. Sie teilt das Verfahren in kleine, übersichtliche Schritte auf:

1. Starten Sie mit der Entwicklung einer Funktion createFrequencies, die die Häufigkeit des Vorkommens eines Zeichen in einem Text berechnet. Verwenden Sie dazu die folgende Signatur:
   `createFrequencies(Text) -> [{Char, Int}]`
2. Entwickeln Sie dann eine Funktion makeLeafList die eine Liste aller Blätter des Baums beinhaltet. Die Liste soll dabei nach den Gewichten der Blätter aufsteigend geordnet sein.
   `makeOrderedLeafList(Freqs: [{Char, Int}]) -> [#leaf]`
3. Schreiben Sie eine Funktion combine die zwei Bäume mit dem geringsten Gewicht der Liste zusammenführt, in dem ein neuer Knoten vom Typ Fork erzeugt wird. Dieser neue Knoten wird in die Knotenliste eingefügt, während die anderen beiden gelöscht werden. Dabei muss die sortierte Reihenfolge eingehalten werden.
   `combine(trees:[fork | leaf]): List[fork |leaf]`
4. Schreiben Sie eine Funktion repeatCombine, die die Funktion combine so lange aufruft, bis nur noch ein Baum in der Liste enthalten ist
   `repeatCombine(trees:[fork | leaf])-> fork |leaf`
5. Entwickeln Sie am Schluss noch die Funktion `createCodeTree` für die Erzeugung des Baums unter Verwendung der von Ihnen geschriebenen Funktionen.

Dekodierung

Definieren Sie eine Funktion decode, die eine Liste von bits mit einem schon definierten Huffman Baum dekodiert.

```
decode(tree: CodeTree, bits: [ 0 | 1] -> [Chars]
```

Kodierung

Definieren Sie eine Funktion, die aus einem Huffman Baum eine Code-Tabelle erzeugt. Diese Tabelle soll sich aus Zweier-Tupeln bestehend aus den Zeichen und dem dazugehörigen Code zusammensetzen.

Entwickeln Sie danach eine Funktion `convert`, die den Huffman-Baum traversiert und dabei eine Code-Tabelle erzeugt.

```
convert(t: CodeTree) -> [{char, BitList}]
```

Schreiben Sie eine als letztes eine Funktion `encode`, die eine Zeichenfolge mit einem Huffman-Baum kodiert.

```
encode(Text , tree: CodeTree)-> [ 0 | 1]
```

Die Belegarbeit kann in Gruppen von 1-2 Personen bearbeitet werden und ist bis zum 01.12.2016 abzugeben. Die Funktionsweise muss in der Übung am 01.12.2016 demonstriert werden. Die Funktionen sollen mindestens die mitgelieferten Tests bestehen. Es ist ratsam, weitere Tests hinzuzufügen, da bei der Kontrolle der Aufgaben evtl. weitere Tests durchgeführt werden.