Tag 8

Beispiel: Tabellen formatieren

Am heutigen Tag geht es nicht in erster Linie darum, neue Konzepte einzuführen, sondern wir wollen sehen, was wir mit dem bereits Erlernten schon erreichen können. Wir wollen Tabellen formatieren, etwa so:

```
> test1 = formatTable [R,L,L]
>            [Row [Entry "Funktionsname"
>                 ,Entry "Typ"
>                 ,Entry "wann"
>                 ]
>            ,Row [Entry "sum"
>                 ,Entry "Num a => [a] -> a"
>                 ,Entry "Tag 4"
>                 ]
>            ,Row [Entry "null"
>                 ,Entry "[a] -> Bool"
>                 ,Entry "Tag 5"
>                 ]
>            ,Row [Entry "foldr"
>                 ,Entry "(a -> b -> b) -> b -> [a] -> b"
>                 ,Entry "noch nicht"
>                 ]
>            ]
```

mit der zugehörigen Ausgabe

```
Tag8> putStrLn test1
Funktionsname  Typ                              wann?
          sum  Num a => [a] -> a                Tag 4
         null  [a] -> Bool                      Tag 5
        foldr  (a -> b -> b) -> b -> [a] -> b   noch nicht
```

Die Funktion `formatTable` bekommt also eine Liste von Spaltenspezifikationen und eine abstrakte Beschreibung der Einträge der Tabelle und macht daraus eine schön formatierte Ausgabe, die die einzelnen Spalten gemäß der Spezifikation links- oder rechtsbündig ausrichtet und die Spaltenbreiten so wählt, daß sie für die längsten Einträge ausreichend platz bieten.
Vorab etwas zur oben erwähnten Funktion `putStrLn`. Diese hat einen interessanten Typ:

```
Tag8> :t putStrLn
String -> IO ()
```

Es handelt sich bei `putStrLn` um eine Ausgabefunktion. Das Argument ist ein `String`, das Resultat eine sogenannte `IO`-Aktion. Wir werden bald mehr über Ein- und Ausgabe lernen, für den Moment genügt es zu wissen, daß man diese Funktion am interaktiven Prompt verwenden kann, um einen `String` *interpretiert* ausgeben zu lassen. Was das heißt, läßt sich am besten an Beispielen

demonstrieren. In Haskell kann man einige Steuerzeichen als Escape-Sequenzen schreiben, zum Beispiel einen Zeilenumbruch als `\n`, einen Tabulator als `\t` oder einen Backspace-Charakter als `\b`.

```
Tag8> "ersteZeile\tund\nzweite Zeile\tEndw\be"
"ersteZeile\tund\nzweite Zeile\tEndw\be"
```

Wie man sieht, werden diese Steuerzeichen von der normalen String-Ausgabefunktion uninterpretiert gelassen, man kann zwar genau sehen, welche Steuerzeichen wo enthalten sind, aber nicht, wie der Text wirklich aussähe. Es findet eben keine „echte“ Ausgabe des Strings statt, sondern vielmehr eine Hilfsanzeige für den Programmierer von Seiten des Interpreters. Eine Verwendung von `putStrLn` hingegen führt wirklich zur Ausgabe des Strings mitsamt Interpretation aller Steuerzeichen:

```
Tag8> putStrLn "ersteZeile\tund\nzweite Zeile\tEndw\be"
erste Zeile     und
zweite Zeile    Ende
```

Da wir Tabellen ausrichten wollen, ist es natürlich hilfreich, so eine Funktion zur Hand zu haben, denn wir wollen die Ergebnisse ja auch in einer „schönen“ Form begutachten können.
Nun aber zum eigentlichen Thema. Wir zerlegen die Aufgabe in mehrere Einzelteile:

- Wir definieren geeignete Datentypen.
- Wir schreiben eine Funktion, die einen String in einer gewissen Breite linksbündig, rechtsbündig oder zentriert ausrichtet.
- Wir schreiben eine Funktion, die die Spaltenbreiten berechnet, die notwendig sind, um die Tabelle erfolgreich auszurichten.
- Schließlich schreiben wir die Funktion, die die Tabelle ausrichtet.

8.1 Datentypen definieren

Wir definieren zunächst einige Datentypen, die für unsere Applikation die Typsicherheit erhöhen. Dadurch, daß wir etwa Tabellenzeilen und -einträge mit gesonderten Typen versehen, erhöhen wir die Lesbarkeit des Codes und auch etwaiger Typfehler für uns.
Wir beginnen mit einem Datentyp, der die drei Möglichkeiten umfaßt, wie eine Tabellenspalte ausgerichtet werden kann: linksbündig, zentriert und rechtsbündig.

```
> data Align = L | C | R
```

Der Datentyp `Align` hat also drei Konstruktoren, nämlich `L`, `C` und `R`. All die Konstruktoren haben keine weiteren Argumente.
Tabelleneinträge sind im Prinzip Zeichenketten, aber wir markieren sie mit einem eigenen Konstruktor `Entry`, um sie von anderen Strings zu unterscheiden:

```
> data Entry = Entry String
```

Der Datentyp `Entry` hat damit nur einen Konstruktor. Dieser heißt ebenfalls `Entry` (diese Namensgleichheit ist in Haskell erlaubt und auch gängige Praxis in solchen Fällen, in denen ein Typ nur einen Konstruktor besitzt). Dieser Konstruktor hat ein Argument vom Typ `String`.
In ähnlicher Weise definieren wir nun Tabellenzeilen. Tabellenzeilen sind Listen von Einträgen, daher definieren wir

```
> data Row = Row [Entry]
```

Das genügt vorerst als Vorbereitung.

8.2 Einzelne Einträge ausrichten

Das Ziel dieses Abschnittes ist es, eine einzelne Zelle, deren Breite wir bereits kennen, auszurichten. Wir wollen eine Funktion

```
> formatEntry :: Align -> Int -> Entry -> String
```

schreiben, die mit Hilfe einer Ausrichtung und einer Breitenangabe (der Typ `Int` ist ein Typ für ganze Zahlen mit Größenbegrenzung; wir verwenden ihn dann, wenn wir keine extrem großen Zahlen erwarten, weil er effizienter ist) aus einem Tabelleneintrag eine Zeichenkette eben dieser Breite erstellt, in der der Tabelleneintrag gemäß der gegebenen Ausrichtung formatiert ist. Beispiel:

```
Tag8> formatEntry C 19 (Entry "Eintrag")
"      Eintrag       "
```

Da wir die Breite später vorab berechnen werden, und zwar ausreichend anhand der gesamten Tabelleneinträge für eine Spalte, ist es nicht erforderlich, daß `formatEntry` korrekt arbeitet, falls die gegebene Breite nicht groß genug für den Eintrag ist.
Offenbar müssen wir, um unser Ziel zu erreichen, den Eintrag mit einer gewissen Anzahl von Leerzeichen auffüllen, wobei die Anzahl der Leerzeichen aus der Länge des Eintrages und der gegebenen Breite zu errechnen ist.
Die Funktion `length` kennen wir bereits zum Berechnen der Länge einer Liste (und damit eines Strings). Hilfreich wäre es, wenn wir eine Funktion

```
> spaces :: Int -> String
```

hätten, die, gegeben eine Zahl n, eine Zeichenkette mit n Leerzeichen liefert, etwa

```
Tag8> spaces 5
"     "
```

Das sollte mit unseren Kenntnissen kein Problem mehr sein:

```
< spaces 0 = ""
< spaces n = ' ' : spaces (n-1)
```

Wie so oft, verwenden wir Rekursion in der Definition der Funktion. Eine Folge von 0 Leerzeichen entspricht der leeren Zeichenkette. Eine Folge von n Leerzeichen entsteht daraus, daß wir an $n-1$ Leerzeichen ein weiteres anhängen, mittels Cons-Operator. Wir verwenden „pattern-matching“ für das erste Argument. Der erste Fall trifft zu, wenn wir `spaces` auf `0` anwenden. Der zweite Fall trifft folglich nur dann zu, wenn wir `spaces` auf einen `Int` ungleich `0` anwenden. Die Funktion `spaces` terminiert nicht, wenn wir sie auf eine negative Zahl anwenden. Wir werden in einer der folgenden Lektionen noch lernen, wie man in einem solchen Fall eine aussagekräftige Fehlermeldung produzieren kann.
Tatsächlich bietet die Haskell-Prelude eine allgemeinere Funktion `replicate` mit dem Typ `Int -> a -> [a]`, welche aus einer Zahl n und einem einzelnen Zeichen die Zeichenkette bildet, die n Mal das betreffende Zeichen enthält.
Wir können `spaces` daher alternativ wie folgt definieren

```
> spaces n = replicate n ' '
```

Aufgabe 1

Schreibe Deine eigene Version `replicate'` von `replicate`.

Es ist nun nicht mehr schwierig, `formatEntry` zu definieren. Wir beginnen mit einem ersten Anlauf für die zentrierte Formatierung:

```
< formatEntry C width (Entry entry) =
<        spaces (div (width - length entry) 2)
<     ++ entry
<     ++ spaces (div (width - length entry + 1) 2)
```

Zunächst verwenden wir „pattern matching“ für die Argumente: Das erste Muster ist `C`, das heißt, der betreffende Fall, den wir soeben geschrieben haben, trifft nur dann zu, wenn das erste Argument auch `C` ist. Das zweite Muster ist einfach ein Name, hier passiert also nichts weiter, als daß der `Int`-Parameter an den Namen `width` gebunden wird. Das dritte Argument ist vom Typ `Entry`. Einträge sind – so wie wir sie definiert haben – immer Zeichenketten, die mit dem Konstruktor `Entry` markiert werden. Durch das Muster `Entry entry` sorgen wir dafür, daß wir die Zeichenkette wieder aus dem Eintrag extrahieren und daß der Name `entry` an diese Zeichenkette gebunden wird.
Das Resultat besteht aus der Konkatenation (unter Verwendung des bereits bekannten Operators `++`) von drei Zeichenketten. Die mittlere ist der Eintrag selbst, die beiden anderen sind Folgen von Leerzeichen, die mit der Funktion `spaces` erzeugt werden. Insgesamt müssen wir `width - length entry` Leerzeichen einfügen, um auf die Breite `width` zu kommen. Diese Anzahl teilen wir in etwa zwei gleich grosse Teile und fügen diese Teile vorne und hinten an den Eintrag an. Für die Teilung verwenden wir dabei den Operator

```
< div :: Integral a => a -> a -> a
```

Dieser führt eine Ganzzahldivision durch (also mit Abrunden). Es gilt für alle `n` vom Typ `Int`, daß

```
div n 2 + div (n+1) 2 == n
```

Natürlich ist es unschön, daß der Teilausdruck `width - length entry` in der Definition von `formatEntry` zweimal vorkommt. Das erhöht nicht nur den Schreibaufwand – es ist auch ineffizient, denn normalerweise wird ein Compiler diesen Ausdruck auch zweimal auswerten (der GHC unterstüzt eine Optimierung, die „common subexpression elimination“ heißt, aber der Aufwand für die Erkennung solcher gemeinsamen Teilausdrücke ist sehr hoch, so daß sie nur in sehr seltenen Fällen funktioniert).
Besser ist es daher, den gemeinsamen Teilaudruck selbst zu identifizieren und ihm in einem `let`-Konstrukt einen temporären Namen zu geben. Das `let`-Konstrukt ist für uns neu und wird hier wie folgt benutzt:

```
> formatEntry C width (Entry entry) =
>     let
>       { n = width - length entry }
>     in
>       spaces (div n 2) ++ entry ++ spaces (div (n+1) 2)
```

Zwischen den Schlüsselwörten `let` und `in` können mehrere lokale Funktionsdefinitionen stehen, mit derselben Syntax, wie wir auch globale (also modulweite) Funktionen definieren. In diesem Fall definieren wir nur eine einzelne Konstante (also eine Funktion ohne Argumente), nämlich `n`, die wir dann in dem Ausdruck, der auf das `in` folgt, benutzen dürfen.

Aufgabe 2

Definiere die beiden fehlenden Fälle von `formatEntry` für das links- und rechtsbündige Ausrichten eines einzelnen Eintrags.

8.3 Spaltenbreiten berechnen

In diesem Abschnitt wollen wir eine Funktion

```
< columnWidths :: [Row] -> [Int]
```

schreiben, die aus einer Liste mit Tabellenzeilen eine Liste mit Spaltenbreiten berechnet. Zunächst ist festzuhalten, daß die Ausrichtung auf die Spaltenbreiten keinen Einflüß hat. Je nach Ausrichtung werden nur Leerzeichen an verschiedenen Stellen aufgefüllt, allein der breiteste Eintrag pro Spalte bestimmt die notwendige Breite einer Spalte. Wir gehen der Einfachheit halber davon aus, daß jede Zeile gleich viele Einträge enthält (etwa n viele). In diesem Fall wird die Ergebnisliste also auch n Spaltenbreiten enthalten.

Daß die Einträge der Tabelle in der Eingabe nach Zeilen gruppiert sind, ist für unsere Aufgabe eher hinderlich. Daher wird ein erstes Teilziel sein, die Tabelle zu „transponieren“, also nach Spalten statt nach Zeilen zu gruppieren. Um dem pädagogischen Anspruch gerecht zu werden, definieren wir uns dafür noch einen Datentyp:

```
> data Column = Column [Entry]
```

Die Funktion, die die Transposition durchführt, hat dann folgenden Typ:

```
> transposeTable                  :: [Row] -> [Column]
```

Auch hier verwenden wir wieder Rekursion, und wir beginnen mit zwei einfachen Basisfällen:

```
> transposeTable []               =  []
> transposeTable [Row es]         =  map (\e -> Column [e]) es
```

Eine Tabelle ganz ohne Zeilen ist natürlich auch eine Tabelle ganz ohne Spalten (diesen Fall könnten wir auch ausschließen).
In der zweiten Zeile sehen wir ein sogenanntes „nested pattern“, was von der bisherigen Form eines Konstruktors, gefolgt von Variablennamen, abweicht. Das Muster trifft genau dann zu, wenn die Liste von Zeilen genau ein Element enthält. Dieses Element (vom Typ `Row`) muß mit dem Konstruktor `Row` gebildet worden sein (was immer der Fall ist), und das Feld des Konstruktors (vom Typ `[Entry]`) wird an den Namen `es` gebunden.
Eine Tabelle mit genau einer Zeile läßt sich transponieren, indem wir für jeden Eintrag eine Spalte erstellen, die genau einen Eintrag enthält. Dazu machen wir Gebrauch von `map`. Zur Erinnerung: Die Funktion `map` nimmt eine Funktion und eine Liste und wendet die Funktion auf jedes Element der Liste an. In unserem Fall haben wir eine Liste von Einträgen (den Einträgen unserer einen Zeile in der Tabelle), und die Funktion, die angewendet werden soll, ist durch einen Lambda-Ausdruck gegeben: Wir nehmen einen (also jeden) Eintrag `e` und bilden ihn ab auf eine Spalte, die genau diesen Eintrag enthält. Fertig!
Der noch ausstehende allgemeine Fall ist das komplizierteste Stückchen Code in der gesamten Lektion, aber wir werden das Schritt für Schritt erklären. Hier zunächst die Definition:

```
> transposeTable (Row es : rows)   =
>   let
>     { rec                        :: [Column]
>     ; rec                        =  transposeTable rows
>     ; addToColumn                :: Entry -> Column -> Column
>     ; addToColumn e (Column es)  =  Column (e : es)
>     }
>   in
>     zipWith addToColumn es rec
```

Auch hier wieder ein „nested pattern“. Das Muster trifft zu, wenn die Liste mittels Cons-Operator gebildet wurde (also mindestens ein Element enthält). Tatsächlich wird die Liste immer mindestens zwei Elemente enthalten, weil im Fall der einelementigen Liste bereits der oben definierte

Basisfall zutrifft. Der Kopf der Liste muß von der Form `Row es` sein, der Rest der Liste wird an `rows` gebunden.
Wir begegnen dem `let`-Konstrukt erneut. Auch hier wäre es – wie bei der ersten Verwendung – möglich, ohne `let` auszukommen. Die definierten Werte könnten im Ausdruck nach dem `in` einfach durch ihre Definitionen ersetzt werden (bei der Funktion `addToColumn`) mit Hilfe eines Lambda-Ausdrucks. Die Einführung von temporären Namen dient hier der Zerlegung des Ausdrucks in kleinere, handlichere Bestandteile, denen wir zu Dokumentationszwecken auch nochmals Typsignaturen verpassen (das ist, wie (fast) immer, nicht nötig; würden die Typsignaturen fehlen, dann würden sie automatisch vom Compiler inferiert).
Die Idee besteht darin, erst alle bis auf die erste Zeile durch einen rekursiven Aufruf der Funktion `transposeTable` zu transponieren. Da die Restliste noch mindestens eine Zeile enthält, ist das Resultat `rec` eine nicht-leere Liste von Spalten. Den Spalten fehlen jedoch ihre jeweils ersten Einträge, denn diese sind in der ersten Zeile (deren Einträge in `es` zwischengespeichert sind) definiert! Das Hinzufügen eines ersten Eintrages zu einer Spalte leistet die Funktion `addToColumn` durch Verwendung des Cons-Operators. Wir haben nun zwei Listen: eine enthält die Einträge der ersten Zeile, die andere enthält die Spalten, denen die ersten Einträge fehlen. Beide Listen haben gleich viele Elemente, und das Resultat, an dem wir interessiert sind, entsteht dadurch, daß beide Listen in eine Liste zusammengeführt werden, und zwar, in dem die jeweils zueinandergehörenden Elemente der beiden Liste mit Hilfe der Funktion `addToColumn` kombiniert werden.
Das genau leistest die Funktion `zipWith`, die in der Haskell-Prelude etwa wie folgt definiert ist:

```
< zipWith                       :: (a -> b -> c) -> [a] -> [b] -> [c]
< zipWith combine []     ys     =  []
< zipWith combine xs     []     =  []
< zipWith combine (x:xs) (y:ys) =  combine x y : zipWith combine xs ys
```

Die ersten beiden Fälle sorgen dafür, daß beim Zusammenfügen zweier Listen unterschiedlicher Länge die längere Liste abgeschnitten wird. Sobald eine der beiden Listen leer ist, ist das Resultat die leere Liste. Wenn beide Listen noch mindestens ein Element enthalten, dann trifft die dritte Zeile zu. In diesem Fall werden die Kopfelemente der beiden Liste mit der Funktion `combine`, dem ersten Argument der Funktion `zipWith` kombiniert und vorne an die rekursiv gebildete Restliste angehängt.
Die maximale Länge eines Spalteneintrags ist nun leicht gefunden:

```
> maximumColumn :: Column -> Int
> maximumColumn (Column es) =
>    let
>      { lengths :: [Int]
>      ; lengths =  map (\(Entry e) -> length e) es
>      }
>    in
>      maximum lengths
```

Zunächst bestimmen wir die Längen der einzelnen Einträge mit Hilfe von `map`. Dann benutzen wir die bereits früher erwähnte Funktion `maximum`, um das Maximum einer Liste zu bestimmen. Damit sind wir für `columnWidths` ausreichend mit Hilfsmitteln bestückt:

```
> columnWidths :: [Row] -> [Int]
> columnWidths rows = map maximumColumn (transposeTable rows)
```

Die Arbeitsweise dieser Funktion sollte mittlerweile ohne weitere Eklärung verständlich sein.

8.4 Tabellen formatieren

Nun ist alle Vorarbeit geleistet, um die eigentliche Formatierung vornehmen zu können. Wir schreiben zunächst noch eine Funktion

```
> formatRow :: [Align] -> [Int] -> Row -> String
```

die eine einzelne Zeile formatiert. Hier haben wir ein ähnliches Problem wie zuvor bei der Definition von `transposeTable`: Eine Zeile enthält für jede Spalte einen Eintrag. Zusätzlich haben wir für jede Spalte eine Ausrichtug und eine Breite. Wir haben also insgesamt drei Listen derselben Länge. Zudem haben wir bereits eine Funktion, nämlich `formatEntry`, die sich eignet, jeweils eine Ausrichtung, eine Breite und einen Eintrag in eine Zeichenkette zu transformieren. Vorhin hatten wir zwei Listen mit Hilfe von `zipWith` zu einer Liste zusammengefügt. Für drei Listen gibt es die verwandte Funktion `zipWith3`, die ebenfalls in der Prelude definiert ist. Daher:

```
> formatRow aligns widths (Row es) =
>     mergeStrings (zipWith3 formatEntry aligns widths es)
```

Der Aufruf von `zipWith3` liefert uns eine Liste von Strings, die die einzelnen, ausgerichteten Tabelleneinträge für eine Zeile enthalten. Diese müssen nun zu einem einzelnen String zusammengefügt werden. Die einfachste Möglichkeit

```
> mergeStrings :: [String] -> String
> mergeStrings =  concat
```

ist suboptimal, da auf diese Weise die Spalten ohne Zwischenraum aneinandergehängt werden.

Aufgabe 3

Schreibe die Funktion `mergeStrings` so, daß zwischen je zwei Spalten noch zwei Leerzeichen eingefügt werden. Beispiel:

```
Tag8> mergeStrings ["Haskell","B.","Curry"]
"Haskell  B.  Curry"
```

Aufgabe 4

Schreibe die Funktion `formatTable` mit dem Typ `[Align] -> [Row] -> String`. Verwende dabei die Funktion

```
< unlines :: [String] -> String
```

die in der Prelude definiert ist und dazu dient, aus einer Liste von Strings, die jeweils eine Zeile enthalten, einen zusammenhängenden String zu bauen, der Zeilenumbrüche an den entsprechenden Stellen enthält. Teste die Funktion unter Verwendung von `putStrLn` und `test1`. Schreibe einen weiteren Test.

Lernziele Tag 8

- ☐ Um lokale Funktionen oder wiederkehrende lokale Ausdrücke zu definieren, ist das `let`-Konstrukt hilfreich.
- ☐ Auch größere Problemstellungen lassen sich bequem in kleine Bruchstücke zerlegen, die jeweils durch einzelne Funktionen realisiert werden können.
- ☐ Die Prelude definiert eine große Anzahl hilfreicher Funktionen (etwa `map` oder `zipWith`), die man im täglichen Gebrauch oft benötigt.