Universität Bremen

Praktische Informatik 3 WS 16/17

Christoph Lüth
Tobias Brandt
Tristan Bruns
Johannes Ganser
Berthold Hoffmann
Alexander Kurth

2. Übungsblatt

Ausgabe: 01.11.16
Abgabe: 11.11.16, 12:00 Uhr

2.1 *Pizza-Toni liefert nicht.* *10 Punkte*

Pizza und zuckerhaltige Sodagetränke sind neben hinreichender Versorgung mit Koffein die artgerechte Ernährung für Informatiker[1]. Deshalb hat Sie der Inhaber von Toni's Pizza-Dienst, Gianluigi "Toni" Corleone, um Hilfe gebeten.

In seinem kleinen Pizza-Dienst geht alles drunter und drüber. Die Lagerhaltung ist ein einziges Chaos, und oft muss die Zubereitung einer Pizza Quattro Stagioni abgebrochen werden, damit der Küchenjunge Guiseppe noch mal (stilecht mit der Vespa) zum Großhandel rasen kann, um fehlende Paprika zu kaufen. Wir wollen deshalb ein computergestütztes Lagerhaltungssytem und Bestellwesen für Herrn Corleone implementieren.

Pizza	Zutaten	Preis klein	groß
Pizza Margherita	Tomate, Käse	4,50	7,00
Pizza Salami	Tomate, Käse, Salami	5,00	8,00
Pizza Hawai	Tomate, Käse, Ananas, Schinken	5,50	9,00
Pizza Diavolo	Tomate, Käse, Peperoni, Salami	6,00	10,00
Pizza Quattro Stagioni	Tomate, Käse, Salami, Schinken, Paprika, Oliven	6,50	11,00

Abbildung 1: Die Speisekarte von Toni's Pizza-Dienst

Abbildung 1 zeigt die Speisekarte. Wir modellieren zuerst die Pizzen und Zutaten mit zwei Aufzählungen, Pizza und Zutat. Der Datentyp Zutaten modelliert eine *Menge* von Zutaten; die folgende Funktion gibt die Zutaten einer Pizza (nach Abb. 1) zurück:

```
zutaten :: Pizza→ Zutaten
```

Die Lagerhaltung soll das Lager als eine Menge von Zutaten mit jeweils einer Anzahl modellieren. Es sollen folgende Funktionen implementiert werden, welche prüfen, ob eine Zutat oder eine Menge von Zutaten im Lager enthalten sind, oder diese hinzufügen oder entnehmen:

```
anzahlZutat      :: Lager→ Zutat→ Int
zutatenImLager   :: Lager→ Zutaten→ Bool
lieferungZutat   :: Lager→ Zutat→ Lager
lieferung        :: Lager→ Zutaten→ Lager
entferneZutat    :: Lager→ Zutat→ Lager
entferneZutaten  :: Lager→ Zutaten→ Lager
```

Für das Bestellwesen implementieren wir den Datentyp Bestellung als eine Menge von Pizzen, und implementieren die folgenden Funktionen:

```
pizzaVerfuegbar       :: Lager→ Pizza → Bool
bestellungVerfuegbar  :: Lager→ Bestellung→ Bool
pizzaBacken           :: Lager→ Pizza→ Lager
bestellungBacken      :: Lager→ Bestellung→ Lager
```

wobei die ersten beiden prüfen, ob die Zutaten für eine einzelne Pizza bzw. eine Bestellung am Lager verfügbar sind, und die letzten beiden, unter der Vorbedingung dass sie verfügbar sind, die Zutaten für eine Pizza bzw. Bestellung aus dem Lager entfernt (und die Pizzen backt — davon abstrahieren wir hier mal).

[1] Nicht wirklich.

2.2 *Rechtfertigungen* *10 Punkte*

Nach Pizza kommen wir zu einem weiteren Kernthema der Informatik, nämlich Textverarbeitung. Wir sind noch ganz am Anfang, daher implementieren wir erstmal nur Textformatierung, und hier auch nur den einfachen Blocksatz (der gar nicht so einfach ist).

Implementieren Sie eine Funktion

```
justify :: Int→ String→ String
```

welche eine gegebene Zeichenkette so formatiert, dass

- alle Zeilen bis auf die letzte Zeile oder Zeilen mit genau einem Wort die angegebene Länge haben;
- die letzte Zeile höchstens die angegebene Länge hat; und
- das Ergebnis und der Eingabgetext bis auf Leerzeichen und Zeilenumbrüche gleich sind.

Dazu benötigen wir eine Datenstruktur Line, die eine aus Worten (leerzeichenfreie Zeichenketten) bestehende Zeile modelliert:

```
data Line = Line String Line | Empty
```

Für diese Datenstruktur benötigen wir folgende Hilfsfunktionen (siehe auch Hinweise unten):

```
numWords    :: Line→ Int              — Anzahl der Worte in der Zeile
len         :: Line→ Int              — Gesamtlänge aller Worte
app         :: Line→ String→ Line     — hängt ein Wort hinten an eine Zeile
stringToLine :: String→ Line
lineToString :: Line→ String
```

Die letzten beiden werden am besten mit einem Beispiel erläutert:

```
*Justify> stringToLine "Foo! Baz, bar   bar   foo"
Line "Foo!" (Line "Baz," (Line "bar" (Line "bar" (Line "foo" Empty))))
*Justify> lineToString it
"Foo! Baz, bar bar foo"
```

Die Funktion stringToLine ist auf dem auf der Webseite verfügbaren Rahmenwerk für diese Aufgabe vorgegeben.

Eine wichtige Funktion ist folgende, welche eine Zeile auf die gewünschte Breite formatiert :

```
formatLine :: Int→ Line→ String
```

Unter der Vorbedingung len l + numWords l − 1 ≤ w liefert formatLine w l eine Zeichenkette der Länge w, die bis auf Leerzeichen mit lineToString l übereinstimmt, und zwar indem sie die Lücken zwischen den Wörtern in l mit Leerzeichen gleichmäßig auffüllt:

```
*Justify> formatLine 20 (Line "aa" (Line "bb" (Line "cc" (Line "dd" (Line "ee" Empty)))))
"aa  bb  cc  dd  ee"
```

Überschüssige Leerzeichen (hier werden 10 Leerzeichen auf 4 Lücken verteilt) werden in den hinteren Lücken (hier die letzten beiden) aufgefüllt, so dass in allen Lücken die Anzahl der Leerzeichen um höchstens 1 differiert, und für alle Lücken die Lücken davor nie mehr Leerzeichen haben.

Die eigentliche Blocksatzfunktion ist

```
justify :: Int→ String→ String
```

und wird wie folgt implementiert:

1. Zuerst wird der Text in eine Line konvertiert.
2. Wir setzen immer eine *aktuelle Zeile*, beginnend mit dem ersten Wort.
3. Zu dieser aktuellen Zeile fügen wir solange Wörter hinzu, bis durch das Hinzufügen eines Wortes die Zeile länger werden würde als die gewünschte Länge.
4. In diesem Fall setzen wir die aktuelle Zeile (ohne das neue Wort) mit der Funktion formatLine auf die gewünschte Länge, und machen rekursiv mit einer neuen aktuellen Zeile, die genau aus dem neuen Wort besteht, weiter.
5. Wenn kein Wort mehr vorhanden ist, setzen wir die aktuelle Zeile mit lineToString; dieses ist die letzte Zeile, sie wird nicht auf die gewünschte Länge aufgefüllt.

Aus der Beschreibung folgt, dass ein Wort, welches länger ist als die gewünschte Zeilenlänge, alleine in eine überlange Zeile gesetzt wird.

Der beschriebene Algorithmus ist ein sogenannter *greedy algorithm*, und nur eine Möglichkeit, Blocksatz zu implementieren. Er führt nicht immer zum schönsten Ergebnis, ist aber am einfachsten zu implementieren und zu verstehen.

Hinweise: Eine weitere nützliche vordefinierte Funktion ist

```
replicate :: Int→ Char→ String  — replicate n c erzeugt einen String aus n Kopien von c
```

? *Verständnisfragen*

1. Welche zusätzliche Mächtigkeit wird durch Rekursion bei algebraischen Datentypen in der Modellierung erreicht? Was läßt sich mit rekursiven Datentypen modellieren, was sich nicht durch nicht-rekursive Datentypen erreichen läßt?
2. Was ist der Unterschied zwischen Bäumen und Graphen, in Haskell modelliert?
3. Was sind die wesentlichen Gemeinsamkeiten, und was sind die wesentlichen Unterschiede zwischen algebraischen Datentypen in Haskell, und Objekten in Java?