Praktische Informatik 3 WS 04/05

3. Übungsblatt

Ausgabe: 29.11.04
Bearbeitungszeit: Zwei Wochen

Christoph Lüth <cxl>
Shi Hui <shi>
Klaus Lüttich <luettich>
Wolfgang Machert <wmachert>
Christian Maeder <maeder>
Hennes Märtins <maertins>
Kai-Florian Richter <richter>
Dennis Walter <dw>
Diedrich Wolter <dwolter>

5 *Advent, Advent.* *6 Punkte*

Jetzt ist wieder Adventszeit, und das beste daran (neben Glühwein) sind sicherlich die dazugehörigen Kekse. Soziologen der University of Crumbles haben herausgefunden, dass es zwei Arten von Keksessern gibt: die einen mögen ihre Kekssortimente ungeordnet, damit sie bei jedem Griff in die Packung einen anderen Keks zu fassen bekommen. Diese Überraschung erhöht für sie den Genuss. Solche Menschen sind impulsiv, risikofreudig, neigen aber etwas zum Leichtsinn; sie werden in ihrem späteren Leben zum Beispiel Torwart beim SV Werder. Die andere Sorte Keksesser sortiert ihr Kekssortiment lieber nach verfügbaren Sorten, damit sie ihre Lieblingssorte gut rationieren können. Diese Menschen sind solide, risikobewusst, und neigen zu Ängstlichkeit und Übervorsicht; sie werden in ihrem späteren Leben zum Beispiel Verwaltungsfachangestellter der mittleren Laufbahn im öffentlichen Dienst.

In dieser Aufgabe geht es um die zweite Sorte Menschen. Es soll eine Funktion implementiert werden, die ein Kekssortiment sortiert. Die verfügbaren Kekssorten sind Zimtsterne, Spekulatius und Pfeffernüsse. Modellieren Sie diese durch einen Aufzählungstyp `Biscuit`, und eine Dose voller Kekse als eine Liste von Keksen

```
data Box = Box [Biscuit]
```

Da es etwas unappetitlich wäre, wild in den Keksen herumzuwühlen, soll auf die Keksdose nur auf zwei Weisen zugegriffen werden können: zum einen kann man sich anschauen, was für ein Keks an der Stelle `i` liegt, zum anderen kann man den Keks an der `i`-ten Stelle mit dem `i`+1-ten Keks vertauschen. Dieses wird durch zwei Funktionen implementiert, eine dritte gibt die Anzahl der Kekse in der Dose an:

```
look :: Box-> Int-> Biscuit
swap :: Box-> Int-> Box
num  :: Box-> Int
```

Mit Hilfe genau dieser drei Funktionen (keine anderen Zugriffe auf `Box`) implementieren Sie jetzt eine Funktion, die eine beliebige Keksdose so sortiert, dass die Zimtsterne vorne liegen, gefolgt von den Gewürzspekulatius, und die Pfeffernüsse dahinter.

6 *Dot-Com zum letzten* *6 Punkte*

Der Internetboom ist ja jetzt schon seit einigen Jahren vorüber, und den Kater danach haben wir auch überstanden. Inzwischen gibt es wieder leicht zu täuschende, äh, risikofreudige *venture capitalists*, die anderer Leute sauer verdientes Geld gerne für alles ausgeben, was einen Punkt im Namen trägt. Sie wollen von dieser neuerlichen Phantasmagorie profitieren, und gründen eine kleine Firma. Ihr Marktvorteil ist die Verwendung einer Computersprache der neuen Generation, so jedenfalls erzählen Sie ihrem Bankberater, die ungeahnte Produktivität und Codequalität ermöglicht. Sie ahnen es schon — diese Sprache ist Haskell.

In dieser Aufgabe werden wir also eine *text rendering engine* für effizientes *content management* implementieren. Dieses soll Dokumente an Hand einer Vorlage formatieren, und dann im Internet darstellen.

Ein *Dokument* besteht aus einem Titel und einer Folge von Textabschnitten. Ein *Textabschnitt* besteht aus

- einer Überschrift und einer Folge von Textabschnitten, oder
- einem Absatz.

Ein *Absatz* besteht aus

- einer Folge von Texten (Zeichenketten), die jeweils hervorgehoben sein können, oder
- aus einer *Aufzählung* von Absätzen.

Ein *formatierter Text* korrespondiert direkt zur Formatierung in der im Internet verbreiteten Hypertext Markup Language (HTML), und besteht aus einer Folge von formatierten Textelementen. Ein formatiertes Textelement (zusammen mit den dargestellten HTML-Formatierungselementen) ist

- ein Stück einfacher Text;
- ein Stück **fettgedruckter** Text (`<b>...</b>`);
- ein Stück *kursiver* Text (`<i>...</i>`);
- eine Aufzählung von formatierten Texten mit Spiegelstrichen oder numeriert (`<ul><li>...<li>...</ul>`, `<ol><li>...<li>...</ol>`);
- eine Überschrift, Unterüberschrift, Unterunterüberschrift oder Unterunterunterüberschrift (`<h1>...</h1>` bis `<h4>...</h4>`).

Eine *Textvorlage* besteht aus einem Kopf und einem Fuß, die beliebiger formatierter Text sind und jedem Dokument voran- bzw. hintenangestellt werden, sowie einer *Formatierungsvorschrift*, die angibt, ob Hervorhebung durch Fett- oder Kursivschrift erfolgt, und ob Aufzählungen durch Spiegelstriche oder numeriert erfolgen sollen.

Implementieren Sie Datentypen `Document`, `FText` und `Template` für Dokumente, formatierten Text und Textvorlage. Implementieren Sie eine Funktion, die ein Dokument unter Verwendung einer Textvorlage in einen formatierten Text umwandelt:

```
format :: Document-> Template-> FText
```

Der Titel des Dokumentes sollte als Überschrift formatiert werden, und die Überschriften verschachtelter Textabschnitte absteigend als Unter^{n}überschriften (mit $1 \leq n \leq 3$).

Implementieren Sie ferner eine Funktion, welche formatierten Text in ein HTML-Dokument umwandelt:

```
render :: FText -> String
```

Hinweis: Ziel dieser Aufgabe ist die möglichst getreue Modellierung der Datentypen, *nicht* die Darstellung von möglichst ausgefallenem HTML.

7 *Eins, Zwei, Drei.* *8 Punkte*

Der Spielemarkt boomt, und wir wollen uns ein Stück des Kuchens abschneiden. Um nicht in Konflikt mit investigativen Fernsehnachrichtenmagazinen und der Freiwilligen Selbstkontrolle zu geraten, nehmen wir allerdings an dieser Stelle Abstand von der Implementierung von *Doom IV* und widmen uns einem harmloseren Spiel, welches im englischen Sprachraum als Tic-Tac-Toe bekannt ist.

Dieses Spiel wird von zwei Spielern (`X` und `O`) gespielt, die auf einem 3×3-Feld abwechselnd Spielsteine auf nichtbesetzte Felder setzen. Wer zuerst drei Steine in einer Reihe, Zeile oder Diagonalen gesetzt hat, hat gewonnen; wenn das Feld voll ist, ohne dass ein Spieler gewonnen hat, ist das Spiel unentschieden.

Wir modellieren Spieler durch einen Aufzählungstyp `Player`, und den Spielbrettzustand als Datentyp `Board`. Der Spielbrettzustand ist eine 3×3-Matrix, bei der an jeder Stelle (i, j) entweder einer der beiden Spieler einen Stein plaziert hat, oder die Stelle ist leer.

Ein einzelner Zug ist ein Paar (i, j) mit $1 \leq i, j \leq 3$. Wir wollen eine Funktion

```
type Move = (Int, Int)
move :: Board-> Player-> Result
```

implementieren, die bei einem Spielbrettzustand für den Spieler den nächsten bestmöglichen Zug berechnet. Der Datentyp `Result` modelliert das Ergebnis: einen gewinnenden Zug, wenn es einen solchen gibt; ansonsten ein zumindest nicht verlierender Zug, wenn es einen solchen gibt; ansonsten Aufgabe. Vorbedingung für `move` ist, dass überhaupt noch ein Zug möglich ist, d.h. das Brett ist noch nicht voll.

Um einen gewinnenden Zug zu berechnen, prüfen wir alle möglichen Züge, ob sie gewinnen. Ein Spieler gewinnt, wenn er direkt in einer Gewinnsituation ist, oder dem Gegner nur Züge läßt, in denen dieser verliert. Ein Spieler verliert, wenn er nicht in einer Gewinnsituation ist, und der Gegner mindestens einen gewinnenden Zug machen kann. (Der Kenner erspäht die Rekursion). Um den Suchraum einzuschränken, reduzieren wir alle möglichen Züge auf die nicht durch Rotation oder Spiegelung ineinander überführbaren.

Dies ist Version 1.1 vom 2004/12/03 14:21:41.