8. Aufgabenblatt zu Funktionale Programmierung vom 01.12.2004, fällig: 08.12.2004

Themen: *Funktionen auf ganzen Zahlen, Zeichenreihen und Bäumen*

Für dieses Aufgabenblatt sollen Sie Haskell-Rechenvorschriften zur Lösung der unten angegebenen Aufgabenstellungen entwickeln und für die Abgabe in einer Datei namens `Aufgabe8.hs` ablegen. Sie sollen also wieder ein "gewöhnliches" Haskell-Script schreiben. Versehen Sie wie auf den bisherigen Aufgabenblättern alle Funktionen, die Sie zur Lösung brauchen, mit ihren Typdeklarationen und kommentieren Sie Ihre Programme aussagekräftig. Benutzen Sie, wo sinnvoll, Hilfsfunktionen und Konstanten. Im einzelnen sollen Sie folgende Problemstellungen bearbeiten:

1. In der ersten Aufgabe betrachten wir wieder die (-2)-adischen Zahlen von Aufgabenblatt 7. Schreiben Sie eine Haskell-Rechenvorschrift `multMin2` mit der Signatur `multMin2 :: String -> String -> String`, die das Produkt zweier (-2)-adischer Zahlen bestimmt. Rechnen Sie dabei im (-2)-adischen System! Es ist also nicht statthaft, die Argumente in eine andere (Zahlsystem-) Darstellung zu überführen, dort zu rechnen und anschließend wieder das Ergebnis zu konvertieren. Das Ein-/Ausgabeverhalten ist dann wie folgt festgelegt: Sind beide Argumente gültige Repräsentation einer (-2)-adischen Zahl, so hat die Funktion als Ergebnis das (-2)-adische Produkt dieser beiden Zahlen; ansonsten ist das Ergebnis die Zeichenreihe `Ungueltige Eingabe`.

2. In der zweiten Aufgabe betrachten wir den folgenden algebraischen Datentyp für arithmetische Ausdrücke über ganzen Zahlen und (einzeichigen) Variablen(namen):

```
data Expr = Opdi Int       |
            Opdc Char      |
            Add Expr Expr  |
            Sub Expr Expr  |
            Mul Expr Expr  |
            Div Expr Expr
```

   (a) Schreiben Sie eine Haskell-Rechenvorschrift `expCreate` mit der Signatur `expCreate :: String -> Expr`. Die als Argument übergebene Zeichenreihe wird ein vollständig, aber nicht überflüssig geklammerter arithmetischer Ausdruck in üblicher Schreibweise mit Infixoperatoren sein, etwa der Ausdruck `(((2+4)*3)-b)` oder der Ausdruck `((a+2)/((3*4)*b))`, kein wie auch immer korrumpierter Ausdruck. Die Ausgabe soll der entsprechende Ausdruck aus `Expr` sein. Für die beiden obigen Ausdrücke sind dies die Ausdrücke `Sub (Mult (Add (Opdi 2) (Opdi 4)) (Opdi 3)) (Opdc 'b')` und `Div (Add (Opdc 'a') (Opdi 2)) (Mul (Mul (Opdi 3) (Opdi 4)) (Opdc 'b')`.

   (b) Schreiben Sie eine Haskell-Rechenvorschrift `expEqual`, die zwei als Argumente übergebene Ausdrücke auf Gleichheit überprüft. Zwei Ausdrücke heißen gleich, wenn sie dieselbe Knotenzahl und dieselbe Struktur besitzen, also knotenweise mit demselben Konstruktor und im Falle von Operandenknoten zusätzlich denselben ganzzahligen bzw. Buchstabenwert tragen. Die Funktion `expEqual` hat also die Signatur `expEqual :: Expr -> Expr -> Bool`. Sie liefert den Wert

`True`, wenn die beiden Argumente im obigen Sinn gleich sind, sonst `False`. Angewendet auf die beiden Beispielausdrücke aus der vorigen Teilaufgabe liefert die Funktion `expEqual` also den Wert `False`.

(c) Schreiben Sie eine Haskell-Rechenvorschrift `isSubExpr` mit der Signatur `isSubExpr :: Expr -> Expr -> Bool`. Die Funktion `isSubExpr` überprüft, ob der als erstes Argument übergebene Ausdruck als Teilausdruck in dem als zweites Argument übergebenen Ausdruck vorkommt. Falls ja, liefert die Funktion `isSubExpr` den Wahrheitswert `True`, sonst `False`.

(d) Schreiben Sie eine Haskell-Rechenvorschrift `expEval`, die einen Ausdruck so weit wie möglich auswertet, d.h. sind beide Operanden eines Operators ganze Zahlen, so wird der entsprechende Teilausdruck ausgewertet und durch den Wert ersetzt. Die Funktion `expEval` soll die Signatur `expEval :: Expr -> Expr` besitzen. Als Resultat soll die Funktion `expEval` den maximal vereinfachten Ausdruck liefern. Für die Beispielausdrücke aus der vorigen Teilaufgabe sind dies die Ausdrücke `Sub (Opdi 18) (Opdc 'b')` und `Div (Add (Opdc 'a') (Opdi 2)) (Mul (Opdi 12) (Opdc 'b'))`.

3. In dieser Aufgabe greifen wir die Auszeichnungssprache von Aufgabenblatt 7 auf. Zur Erinnerung halten wir noch einmal fest, dass das Klammerpaar { } fett gedruckten Text und das Klammerpaar [ ] normalen Text auszeichnet. Öffnende Klammern schalten jeweils auf den entsprechenden Auszeichnungsmodus um, schließende Klammern beenden ihn. Alle anderen Zeichen haben keine spezielle Bedeutung. Mit diesen Festlegungen lassen sich in Ausdrücken unserer Auszeichnungssprache oft Klammern einsparen, ohne dass sich die Darstellung des Ausdrucks änderte. Beispielsweise führen die folgenden zwei Ausdrücke

```
"[Die Beschreibung] ist {anders}, [[doch die] Darstellung ist {gleich}]!"
"[Die Beschreibung] ist {anders},  [doch die Darstellung ist {gleich}]!"
```

auf dieselbe Darstellung. Tatsächlich gibt es sogar einen Ausdruck minimaler Länge, ohne dass sich die Darstellung änderte:

```
"Die Beschreibung ist {anders}, doch die Darstellung ist {gleich}!"
```

Schreiben Sie nun eine Haskell-Rechenvorschrift `canonicalMarkup`, die einen vorgelegten Ausdruck aus unserer Ausdruckssprache in der Länge minimiert. Für die Minimierung legen wir zusätzlich fest, dass Leerzeichen (ausschließlich "space", keine Tabulatorsprünge, etc.) sowohl fett als auch normal gleich aussehen und (nach der Minimierung) nicht unmittelbar nach einer öffnenden und unmittelbar vor einer schließenden Auszeichnungsklammer stehen.

Die Funktion `canonicalMarkup` hat also die Signatur `canonicalMarkup :: String -> String`. Das Ein-/Ausgabeverhalten ist dabei wie folgt festgelegt: Ist das Argument ein wohlgeformter Ausdruck im Sinne von Aufgabenblatt 7, so ist unter den obigen Randbedingungen das Resultat der kürzeste Ausdruck der Ausdruckssprache, der dieselbe Darstellung besitzt; anderenfalls die Zeichenreihe `Ungueltige Eingabe`.