

Softwaretechnik und Programmiersprachen WS11/12 Haskell-Blatt 2

Aufgabe 1 (Differenz-Listen – sehr leicht)

Man kann in Haskell Differenz-Listen folgendermassen definieren:

```
type DList a = [a] -> [a]
fromDList :: DList a -> [a]
fromDList l = l []
```

Die Funktion `fromDList` konvertiert eine Differenz-Liste in eine normale Liste. Implementieren Sie folgende Funktionen:

```
toDList :: [a] -> DList a
appendDList :: DList a -> DList a -> DList a
consDList :: a -> DList a -> DList a
snocDList :: DList a -> a -> DList a
squareDList :: DList a -> DList a      -- soll \x -> x ++ x entsprechen
cubeDList :: DList a -> DList a      -- soll \x -> x ++ x ++ x entsprechen
```

Hinweis: Nutzen Sie `(.) :: (b -> c) -> (a -> b) -> a -> c` um Differenz-Listen effizient zu machen.

Im Modul `Data.Sequence` ist ein weiterer Datentyp für Sequenzen definiert. Wozu dienen `viewl` und `viewr` aus diesem Modul?

Haskell Differenz-Listen sind persistent. Welchen Vorteil hat dies im Vergleich zu Prolog Differenz-Listen?

Aufgabe 2 (Left-Fold durch Right-Fold ausdrücken – zum knobeln)

Implementieren Sie `foldl` mit Hilfe von `foldr`. Hinweise:

- 1) Es geht nicht darum `foldl` *irgendwie* zu definieren und dabei *irgendwo* `foldr` einzubauen, sondern darum `foldl` auf `foldr` zurückzuführen.
- 2) Überlegen Sie sich zuerst eine (naive/ineffiziente) Lösung.
- 3) Es gibt eine sehr kompakte Musterlösung für diese Aufgabe.
- 4) Wenn Sie nach einigen Stunden keine Lösung gefunden haben googlen Sie danach.
- 5) Versuchen Sie von der ineffizienten Lösung systematisch zur *Muster-Lösung* zu gelangen. (z.B. von Differenz-Listen inspirieren lassen) oder versuchen Sie die Musterlösung zu verstehen.
- 6) Die Aufgabe ist zum Knobeln! Bitte nicht durch solche Knobel-Aufgaben von Haskell *abschrecken* lassen. Die Aufgabe hat mit Haskell nichts zu tun ausser, dass man mit Haskell die Lösung gut hinschreiben/erarbeiten kann.

Aufgabe 3 (Neue Monaden definieren – mittel schwer – zum üben)

In dieser Aufgabe sollen Sie zwei neue Monaden definieren. In der Vorlesungen wurden zwei (separate) Monaden vorgestellt, welche folgende Datentypen benutzen:

```
data MyException b a = OK a | Fehler b
newtype NDet a = NDet {unDet :: [a]}
```

MyException unterstützt Ausnahmen (durch myThrow und myCatch) und NDet unterstützt nicht-Determinismus (durch mzero und mplus).

Es gibt zwei (naheliegende) Möglichkeiten nicht-Determinismus mit Ausnahmen zu verbinden:

1. Möglichkeit: Eine Berechnung liefert entweder eine Liste von Ergebnissen oder genau eine Ausnahme (ExNDet).
2. Möglichkeit: Eine Berechnung liefert eine Liste von Alternativen. Jede Alternative ist entweder ein Ergebnis oder eine Ausnahme (NDetEx).

Implementieren Sie beide Möglichkeiten. Im einzelnen implementieren Sie:

```
data ExNDet = ...
instance Monad ExNDet where ...
instance MonadPlus ExNDet where ...
throwExNDet = ...
catchExNDet = ...
```

```
data NDetEx = ...
instance Monad NDetEx where ...
instance MonadPlus NDetEx where ...
throwNDetEx = ...
catchNDetEx = ...
```

Aufgabe 4 (Parser-Kombinatoren – nicht schwer)

Auf der Webseite finden Sie den Beispiel-Parser `Parser.hs`.

Dokumentation zum Modul `Text.ParserCombinators.Parsec` findet man im Internet.

1) Erweitern Sie `Parser.hs` um for-loops.

Siehe folgende Blöcke in do-Notation:

```
b1= do { lexSym "-" ; e <- baseExp ; return $ Neg e}
b2= do { i <- intLit; return $ Const i}
b3= do { i <- parseIdent; return $ Ident i}
b4= do { lexKey "print"; e <- parseExp; return $ Print e}
b5 = do { lexKey "while"; e <- parseExp; b1 <- parseSBlock; return $ While e b1}
```

2) Schreiben Sie die Blöcke mit Hilfe von `>>=` und `>>`.

3) Schreiben Sie `b2` und `b3` mit `fmap`.

4) Schreiben Sie `b5` mit Hilfe von `<*>` und `<$>` aus `Control.Applicative`.

5) Schreiben Sie `b5` mit Hilfe von `liftM2`.

6) Ersetzen Sie alle vorkommen von `<|>` in der Definition von `parseStmt` durch `Text.Parsec.Combinator.choice`.

7 Implementieren Sie zwei Interpreter für den `Exp` Datentyp (den `Ident` -Fall können Sie ignorieren).

7 a) Einen Interpreter vom Typ `Exp -> Integer`

7 b) Einen Interpreter vom Typ `Monad m => Exp -> m Integer`