6. Aufgabenblatt zu Funktionale Programmierung vom 23.11.2016.
Fällig: 30.11.2016 (15:00 Uhr)

Themen: *Funktionen auf Stapeln und arithmetischen Ausdrücken*

Für dieses Aufgabenblatt sollen Sie Haskell-Rechenvorschriften für die Lösung der unten angegebenen Aufgabenstellungen entwickeln und für die Abgabe in einer Datei namens `Aufgabe6.hs` ablegen. Sie sollen für die Lösung dieses Aufgabenblatts also ein "gewöhnliches" Haskell-Skript schreiben. Versehen Sie wieder 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.

Zur Frist der Zweitabgabe: Siehe allgemeine Hinweise auf Aufgabenblatt 1.

1. Wir betrachten folgenden Typ zur Darstellung von Stapeln:

   ```
   data Stack a = Stk [a] | NoStk deriving (Eq,Show)
   ```

   Dabei repräsentiert `NoStk` keinen gültigen Stack-Wert, sondern dient als Fehlerwert. Zugriff auf und Modifikation von Stapelwerten erfolgt ausschließlich mithilfe der auf Stack-Werten zur Verfügung stehenden Funktionen (siehe unten), die ein *LIFO* (*last in, first out*) Verhalten für Stapel realisieren.

   Zusätzlich zu `Stack a` betrachten wir folgenden Typ:

   ```
   data Maybe a = Just a | Nothing deriving (Eq,Show)
   ```

   Schreiben Sie Haskell-Rechenvorschriften

   ```
   empty   :: (Eq a,Show a) => Stack a
   isEmpty :: (Eq a,Show a) => Stack a -> Bool
   top1    :: (Eq a,Show a) => Stack a -> a
   top2    :: (Eq a,Show a) => Stack a -> Maybe a
   pop     :: (Eq a,Show a) => Stack a -> Stack a
   push    :: (Eq a,Show a) => a -> Stack a -> Stack a
   ```

   zur Manipulation von Werten des Typs `Stack a`.

   - Die 0-stellige Funktion `empty` erzeugt den leeren Stack repräsentiert durch den Wert `Stk []`.
   - Angewendet auf den leeren Stack `Stk []`, liefert die Wahrheitswertfunktion `isEmpty` den Wert `True`. Angewendet auf einen nichtleeren Stack `Stk (l:ls)` oder den Fehlerwert `NoStk`, liefert sie den Wert `False`.
   - Angewendet auf einen nichtleeren Stack s, liefert die Funktion `top1` den Wert des zuletzt auf s abgelegten Eintrags; angewendet auf den leeren Stack oder den Fehlerwert `NoStk`, bricht die Funktion `top1` mit dem Aufruf `error "Invalid Argument"` ab.
   - Angewendet auf einen nichtleeren Stack s, liefert die Funktion `top2` den Wert `Just e` des zuletzt auf s abgelegten Eintrags e; angewendet auf den leeren Stack oder den Fehlerwert `NoStk`, liefert die Funktion `top2` den Wert `Nothing`.
   - Angewendet auf einen nichtleeren Stack s, entfernt die Funktion `pop` den zuletzt in s eingefügten Eintrag und liefert den so verkürzten Stack als Resultat; angewendet auf den leeren Stack oder den Fehlerwert `NoStk`, liefert sie den Wert `NoStk`.
   - Angewendet auf einen `a`-Wert e und einen von `NoStk` verschiedenen Stack s, legt die Funktion `push` den Wert e als neuen, zuletzt eingefügten Eintrag auf s ab und liefert diesen verlängerten Stack als Resultat ab; angewendet auf einen `a`-Wert e und den Fehlerwert `NoStk`, liefert sie `NoStk` als Resultat.

2. Wir betrachten noch einmal den Datentyp der Numerale zur Darstellung ganzer Zahlen zur Basis 3 von Aufgabenblatt 3 zusammen mit allen dort eingeführten Angaben zu Kanonizität, etc. Zusätzlich führen wir folgende weitere Datentypen ein:

   - Einen Datentyp `Expression` zur Darstellung arithmetischer Ausdrücke über Numeralen und einer dreielementigen Menge von Variablen in Postfixnotation (gleichbedeutend mit umgekehrt Polnischer Notation). Der Ausdruck $((a + b) * c)$ in Infixnotation entspricht dem wertgleichen Ausdruck $a\,b\,+\,c*$ in Postfixnotation.
   - Einen Datentyp `Variable` als Darstellung einer 3-elementigen Variablenmenge.
   - Einen Datentyp `State` zur Darstellung von Zuständen. Ein Zustand ist eine Belegung von Variablen mit (Numeral-) Werten.

```
data Digit      = Zero | One | Two deriving (Eq,Enum,Show)
type Digits     = [Digit]
data Sign       = Pos | Neg  deriving (Eq,Show) -- Pos fuer Positive,
                                                -- Neg fuer Negative
newtype Numeral = Num (Sign,Digits) deriving (Eq,Show)

data Operator   = Plus | Times | Minus deriving (Eq,Show)
                                                -- Plus fuer Addition,
                                                -- Times fuer Multiplikation,
                                                -- Minus fuer Subtraktion
data Variable   = A | B | C deriving (Eq,Show)
data Expression = Cst Numeral
                  | Var Variable
                  | Exp Expression Expression Operator deriving (Eq,Show)
type State      = Variable -> Numeral  -- Total definiert, Numeralwert
                                       -- gueltig und kanonisch
```

   Schreiben Sie eine Haskell-Rechenvorschrift

```
eval :: Expression -> State -> Integer
```

   zur Auswertung von Ausdrücken e in Zuständen s. Dabei wird der Wert des Ausdrucks als Dezimalwert vom Typ `Integer` zurückgegeben. Sie dürfen davon ausgehen, dass Zustände stets total definiert sind und Variablen stets mit gültigen Numeralwerten in kanonischer Darstellung belegt sind.

   Angewendet auf einen Ausdruck e und einen Zustand s, liefert die Funktion `eval` den Dezimalwert von e im Zustand s. Dabei gilt: Der Wert einer Konstanten ist (unabhängig vom Zustand) der Dezimalwert des Numeralwerts zur Basis 3 der Konstanten, der Wert einer Variablen im Zustand s ist der Dezimalwert des Numeralwerts zur Basis 3 der Variablen im Zustand s. Der Wert zusammengesetzter Ausdrücke mit Operatoren ergibt sich in rekursiver Weise, wobei die Operatoren `Plus, Times, Minus` als Addition, Multiplikation und Subtraktion interpretiert werden.

   Anwendungsbeispiel:

```
e1 = Exp (Cst (Num (Pos,[One,Zero,Two]))) (Var B) Plus -- entspricht (11+B)
e2 = Exp (Exp (Cst (Num (Neg,[One,Zero,Two]))) (Var B) Plus) (Var C) Minus -- ((-11)+B)-C)
e3 = Exp (Exp (Cst (Num (Pos,[One,Zero,Two]))) (Var B) Minus) (Var C) Times) -- ((11-B)*C)
s :: State
s A = Num (Pos,[Zero])         -- entspricht s(A) = 0
s B = Num (Neg,[One,Zero,One]) -- s(B) = -10
s C = Num (Pos,[One,One,One])  -- s(C) = 13
eval e1 s ->> 1
eval e2 s ->> -34
eval e3 s ->> 273
```

3. Wir betrachten eine alternative Darstellung von Ausdrücken in Form von Listen von Konstanten, Variablen und Operatoren und einen modifizierten Zustandsbegriff:

```
data CVO    = Cop Numeral -- Konstantenoperand
              | Vop Char -- Variablenoperand
              | OpPlus   -- Operator fuer Addition
              | OpTimes  -- Operator fuer Multiplikation
              | OpMinus  -- Operator fuer Subtraktion
                deriving (Eq,Show)
type Expr   = [CVO]
type State2 = Char -> Numeral -- Total definiert, Numeralwert
                              -- gueltig und kanonisch
```

Schreiben Sie eine Haskell-Rechenvorschrift

```
eval2 :: Expr -> State2 -> Integer
```

die wie `eval` den Wert eines Ausdrucks als Dezimalwert vom Typ `Integer` zurückgibt. Für die Implementierung dürfen Sie davon ausgehen, dass `eval2` ausschließlich mit `Expr`-Werten aufgerufen wird, die gültige Postfix-Darstellungen eines arithmetischen Ausdrucks sind und dass Zustände stets total definiert sind und die Variablen eines Zustands stets mit gültigen Numeralwerten in kanonischer Darstellung belegt sind.

Anwendungsbeispiel:

```
s2 :: State2
s2 'A' = Num (Pos,[Zero])         -- entspricht s2('A') = 0
s2 'B' = Num (Neg,[One,Zero,One]) -- s2('B') = -10
s2 'C' = Num (Pos,[One,One,One])  -- s2('C') = 13
s2 _   = Num (Pos,[One])
e = [Vop 'A',Vop 'B',OpMinus,Vop 'C',OpPlus]
eval2 e s2 ->> 23
```

Hinweis: Überlegen Sie sich, ob Ihnen für die Auswertung von Ausdrücken gemäß `eval` oder `eval2` Stapel zur Ablegung von Ausdrücken und Zwischenergebnissen und zur Rechnung hilfreich sein können.

Wichtig: Wenn Sie einzelne Rechenvorschriften aus früheren Lösungen für dieses oder spätere Aufgabenblätter wieder verwenden möchten, so kopieren Sie diese in die neue Abgabedatei ein. Ein `import` schlägt für die Auswertung durch das Abgabeskript fehl, weil Ihre frühere Lösung, aus der importiert wird, nicht mit abgesammelt wird. Deshalb: Kopieren statt importieren zur Wiederwendung!

Haskell Live

Der nächste *Haskell Live*-Termin findet am Freitag, den 25.11.2016, statt.