8. Übungsaufgabe zu
Fortgeschrittene funktionale Programmierung
Thema: Logische Programmierung funktional
ausgegeben: Di, 13.06.2017, fällig: Di, 20.06.2017

Für dieses Aufgabenblatt sollen Sie Haskell-Rechenvorschriften zur Lösung der im folgenden angegebenen Aufgabenstellungen entwickeln und für die Abgabe in einer Datei namens `AufgabeFFP8.hs` auf **oberstem Niveau in Ihrem Gruppenverzeichnis** ablegen. Kommentieren Sie Ihre Programme aussagekräftig und benutzen Sie, wo sinnvoll, Hilfsfunktionen und Konstanten.

- Implementieren Sie die Kombinatorbibliothek aus Kapitel 14 der Vorlesung (siehe auch die unter Anmerkungen beigefügte Datei mit einem Großteil des Codes aus Kapitel 14; siehe auch die aktualisierte Fassung von Kapitel 14 im Foliensatz vom 13.06.2017).
- Ergänzen Sie fehlende Implementierungen, insbesondere die Instanzbildungen für die Typen `Term`, `Subst`, und `Answer` für die Typklasse `Show`, um Ausgaben analog zu den Beispielen aus Kapitel 14 zu ermöglichen.
- Überprüfen Sie Ihre Implementierung anhand der Prädikate `appennd` und `good` aus Kapitel 14 der Vorlesung anhand von Aufrufen der Form:

```
run (append (list [1,2], list [3,4], var "z")) :: Stream Answer
 ->> [{z=[1,2,3,4]}]
run (append (var "x", var "y", list [1,2,3])) :: Stream Answer
 ->> [{x = Nil, y = [1,2,3]},
      {x = [1], y = [2,3]},
      {x = [1,2], y = [3]},
      {x = [1,2,3], y = Nil}]
run (append (var "x", list [2,3], list [1,2,3])) :: Stream Answer
 ->> [{x = [1]}]
run (good (list [1,0,1,1,0,0,1,0,0])) :: Stream Answer
 ->> [{}]
run (good (list [1,0,1,1,0,0,1,0,1])) :: Stream Answer
 ->> []
run (good (var "s")) :: Stream Answer
  ->>[{s=[0]},
      {s=[1,0,0]},
      {s=[1,0,1,0,0]},
      {s=[1,0,1,0,1,0,0]},
      {s=[1,0,1,0,1,0,1,0,0]}
```

```
run (good (var "s")) :: Diag Answer
  ->> Diag [{s=[0]},
            {s=[1,0,0]},
            {s=[1,0,1,0,0]},
            {s=[1,0,1,0,1,0,0]},
            {s=[1,1,0,0,0]},
            {s=[1,0,1,0,1,0,1,0,0]},
            {s=[1,1,0,0,1,0,0]},
            {s=[1,0,1,1,0,0,0]},
            {s=[1,1,0,0,1,0,1,0,0]}
run (good (var "s")) :: Matrix Answer
 ->>MkMatrix [[],
       [{s=[0]}],[],[],[],
       [{s=[1,0,0]}],[],[],[],
       [{s=[1,0,1,0,0]}],[],
       [{s=[1,1,0,0,0]}],[],
       [{s=[1,0,1,0,1,0,0]}],[],
       [{s=[1,0,1,1,0,0,0]},{s=[1,1,0,0,1,0,0]}],[], (usw.)
```

- Entwickeln Sie analog zum Prädikat `append` aus der Vorlesung ein Prädikat

  ```
  shuffleLP :: Bunch m => (Term, Term, Term) -> Pred m
  shuffleLP (p,q,r) = ...
  ```

  `shuffleLP` (LP wie Logikprogrammierung) soll sich für `Int`-Listen wie die Funktion `shuffle` verhalten (s.u.), jedoch soll wie bei `append` die Unterscheidung von Ein- und Ausgabevariablen aufgehoben sein.

  ```
  shuffle :: [a] -> [a] -> [a]
  shuffle [] ys     = ys
  shuffle (x:xs) ys = x : shuffle ys xs
  ```

  Testen Sie Ihr Prädikat `shuffleLP` mit geeigneten Aufrufen, z.B.:

  ```
  run (shuffleLP (list [1,2,3], list [4,5,6], var "z")) :: Stream Answer
   ->> [{z=[1,4,2,5,3,6]}]
  run (shuffleLP (var "x", list [4,5,6], list [1,4,2,5,3,6])) :: Stream Answer
   ->> [{x=[1,2,3]}]
  run (shuffleLP (var "x", var "y", list [1,3,2,4])) :: Stream Answer
   ->> ...viele Belegungsmoeglichkeiten fuer x und y
  ```

  Untersuchen Sie, wie sich die Änderung der Suchmonade (d.h. `Diag Answer` oder `Matrix Answer` statt `Stream Answer`) auf die Ausgaben auswirkt.

- Wir betrachten regulär-artige Sequenzen über den (atomaren) Listen [1], [2] und [3]:

  1. *Atome*: Die Sequenzen [1], [2] und [3] sind regulär-artig.
  2. *Komposition*: Wenn s_1 und s_2 regulär-artige Sequenzen sind, dann ist auch die Sequenz $s_1 + +s_2$ regulär-artig.
  3. *Alternative*: Wenn s_1 und s_2 regulär-artige Sequenzen sind, dann ist auch die Sequenz $s_1 + +[0] + +s_2$ regulär-artig.
  4. Außer den nach diesen Regeln gebildeten Sequenzen gibt es keine weiteren regulär-ähnlichen Sequenzen.

  Schreiben Sie analog zum Prädikat `good` aus Kapitel 14 der Vorlesung ein Prädikat `regSeq` mit Typ

```
regSeq :: Bunch m => Term -> Pred m
regSeq (s) = ...
```

  zur Erkennung und Generierung regulär-artiger Listen. Testen Sie Ihr Prädikat mit Aufrufen der Form:

```
run (regSeq (list [2,3,0,3,1,0,2])) :: Stream Answer
 ->> [{}]          (mit der Bedeutung: Argument war regulaer-artig)
run (regSeq (list [0,2,3,0,3,0,0,1,2])) :: Stream Answer
 ->> []      (mit der Bedeutung: Argument war nicht regulaer-artig)
run (regSeq (var "r")) :: Stream Answer
 ->> ...
run (regSeq (var "r")) :: Diag Answer
 ->> ...
run (regSeq (var "r")) :: Matrix Answer
 ->> ...
```