Advanced Functional Programming: Assignment 7 (Wed, 06/05/2019)
Topic: Logical Programming Functionally
Submission deadline: Wed, 06/12/2019 (3pm)

Regarding the deadline for the second submission: Please, refer to „Hinweise zu Organisation und Ablauf der Übung“ available at the homepage of the course.

Store all functions to be written for this assignment in a top-level file `assignment7.hs` of your group directory. Comment your program meaningfully; use auxiliary functions and constants, where reasonable.

1. Implement the combinator library of Chapter 14 (see also the file provided in column ‘Remarks’ containing major parts of the code of Chapter 14).
2. Add/complete missing implementations of functions, especially the instance declarations for the types `Term`, `Subst`, and `Answer` for the type class `Show` in order to render possible outputs as shown in the examples of Chapter 14.
3. Test and validate your implementation by means of the predicates `append` and `good` of Chapter 14 using calls like:

```
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.)
```

4. Following the implementation of the predicate `append` of Chapter 14, implement a predicate:

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

   For `Int` lists, the behaviour of `shuffleLP` (LP reminds to logical programming) shall be the same as that of function `shuffle` (see below), however, like for `append` the distinction between input and output variables shall be abolished:

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

   Test your implementation of `shuffleLP` with appropriate calls, e.g.:

   ```
   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
    ->> ...many values possible for x and y
   ```

   Investigate also the impact of changing the search monad (i.e., replacing `Stream Answer` by `Diag Answer` or `Matrix Answer`) on the output.

5. We consider sequences over the (atomic) lists [1], [2], and [3], which resemble regular expressions, and are thus called *regular-like sequences* in the following:

   1. *Atoms*: The sequences [1], [2], and [3] are regular-like.
   2. *Composition*: If s_1 and s_2 are regular-like sequences, then also the sequence $s_1 + +s_2$ is regular-like.
   3. *Alternative*: If s_1 und s_2 are regular-like sequences, then also the sequence $s_1 + +[0] + +s_2$ is regular-like.
   4. Rules 1 to 3 define all regular-like sequences; there are no other ones.

   Following the implementation of the predicate `good` of Chapter 14, implement a predicate `regSeq` of type:

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

   for recognizing and generating regular-like sequences. Test your implementation of `regSeq` with calls like:

```
run (regSeq (list [2,3,0,3,1,0,2])) :: Stream Answer
 ->> [{}]          (w/ the meaning: Argument was regular-like)
run (regSeq (list [0,2,3,0,3,0,0,1,2])) :: Stream Answer
 ->> []      (w/ the meaning: Argument was not regular-like)
run (regSeq (var "r")) :: Stream Answer
 ->> ...
run (regSeq (var "r")) :: Diag Answer
 ->> ...
run (regSeq (var "r")) :: Matrix Answer
 ->> ...
```

Important: *Do not use self-defined modules!* If you want to re-use functions (written for earlier assignments), copy these functions to the new submission file. An `import` declaration for self-defined modules will fail, since only the submission file `assignment`i`.hs` , where i, $1 \leq i \leq 8$ (*tentatively*), denotes the running number of the assignment, willl be copied for the (semi-automatic) evaluation. No other file in addition to `assignment`i`.hs` will be copied.