Assignment 5
Advanced functional Programming
Topic: Logical Programming functionally, Streams
Issued on: 04/24/2009, due date: 05/14/2009

For this assignment a Haskell script named `AssFFP5.hs` shall be written offering functions which solve the problems described below. This file `AssFFP5.hs` shall be stored in the home directory of your individual account (not of your group account), as usual on the top most level. Comment your programs meaningfully. Use constants and auxiliary functions, where appropriate.

- Implement a Haskell function `streamFold` with the signature `streamFold :: (a -> b -> a) -> a -> [b] -> [a]`. Applied to a function `f` of type `a -> b -> a`, a value `w` of type `a` and a stream `s` of type `[b]`, the function `streamFold` shall yield a stream `t` of type `[a]` as result. The first element of this stream shall be `w`, the subsequent elements shall be the result of folding the elements of the corresponding prefix of `s`. E.g.:

  ```
  streamFold (+) 0 [1..] == [0,1,3,6,10,15,...
  streamFold (*) 1 [1..] == [1,1,2,6,24,120,...
  ```

- Two natural numbers greater or equal 1 are called *d-friends*, if their Goedel number differ at most by d, $d \in \mathbb{N}_0$.

  Implement a Haskell function `friends` with the signature `friends :: Integer -> [(Integer,Integer)]`. For non-negative arguments d, the function `friends` shall yield the stream of natural numbers of d-friends; for negative arguments, it shall yield the $(0, 0)$-stream, i.e., the stream of pairs of zeros. Take care that the implementation of the function is fair. To this end think about a suitable diagonalization, which visits the pairs of natural numbers in the following order: $[(1, 1), (2, 1), (1, 2), (3, 1), (2, 2), (1, 3), (4, 1), (3, 2), ...]$.

- Similarly to the function `append` of Lecture 5, implement a Haskell function `inject` with signature `inject :: Bunch m => (Term, Term, Term) -> Pred m`. For lists `a`, `b`, `c` the relation `inject (a,b,c)` shall hold, if `c` is the list, which results from `a` by inserting `b` in its middle. If the length of `a` is odd, then `b` shall be inserted right after the middle element `a`; if the length of `a` is even, `b` shall be inserted in the middle of a. The following examples illustrate the intended behaviour:

```
?run(inject(list [1,2,3], list [4,5], var "z")) :: Stream Answer
[{z=[1,2,4,5,3]}]

?run(inject(list [1,2,3,4], list [5.6], var "z")) :: Stream Answer
[{z=[1,2,5,6,3,4]}]
```

- Analogously to the function good from Lecture 5 develop a Haskell function bad for detecting and generating “bad” sequences of zeros and ones. Bad sequences are defined by the following rules:

  1. The sequence [1] is bad.
  2. If s_1 and s_2 are two bad sequences, then the sequence $s1++s_2++[0]$ is bad, too.
  3. Except of sequences constructed according to rule 1 and 2, there are no other bad sequences.

  Note: Intuitively, bad sequences result from traversals of binary trees in postfix order, where branches are labeled with zeros, leafs with ones.

  Similarly to good, the function bad shall be callable in the following fashion:

```
?run(bad(list [1,1,0])) :: Stream Answer
[{}]

?run(bad(list [0,0,1])) :: Stream Answer
[]

?run(bad(var "s")) :: Stream Answer
[{s=[1]},{s=[1,1,0]},{s=[1,1,0,1,0]},{s=[1,1,0,1,0,1,0]},...

?run(bad(var "s")) :: Diag Answer
[{s=[1]},{s=[1,1,0]},{s=[1,1,0,1,0]},{s=[1,1,1,0,0]},...
```