3. Übungsaufgabe zu
Fortgeschrittene Funktionale Programmierung
Thema: Ströme und logische Programmierung funktional
ausgegeben: 27.05.2006, fällig: 07.06.2006

Für dieses Aufgabenblatt sollen Sie Haskell-Rechenvorschriften zur Lösung der im folgenden angegebenen Aufgabenstellungen entwickeln und für die Abgabe in einer Datei namens `AufgabeFFP3.hs` in Ihrem Homeverzeichnis ablegen, wie gewohnt auf oberstem Niveau. Kommentieren Sie Ihre Programme aussagekräftig und benutzen Sie, wo sinnvoll, Hilfsfunktionen und Konstanten.

- Schreiben Sie eine Haskell-Rechenvorschrift `streamFold` mit der Signatur `streamFold :: (a -> b -> a) -> a -> [b] -> [a]`, die angewendet auf eine Funktion `f` vom Typ `a -> b -> a`, einen Wert `w` vom Typ `a` und einen Strom `s` vom Typ `[b]` einen Strom `t` vom Typ `[a]` als Resultat liefert, der mit `w` beginnt und dessen weitere Elemente das Faltungsresultat des entsprechenden Anfangsstroms sind. Es soll also z.B. gelten:

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

- Zwei natürliche Zahlen größer oder gleich 1 heißen *d-befreundet*, wenn sich ihre Gödelzahlen höchstens um d mit $d \in \mathbb{N}_0$ unterscheiden.

  Schreiben Sie eine Haskell-Rechenvorschrift `friends :: Integer -> [(Integer,Integer)]`, die für nicht-negative Argumente d den Strom d-befreundeter natürlicher Zahlen liefert, für negative Argumente den $(0,0)$-Strom liefert, d.h. den nur aus Paaren von Nullen bestehenden Strom. Achten Sie darauf, dass ihre Funktion fair ist und überlegen Sie sich dazu eine Diagonalisierung, die die Paare natürlicher Zahlen nach folgendem Muster durchäuft: $[(1,1),(2,1),(1,2),(3,1),(2,2),(1,3),(4,1),(3,2),...]$.

- Entwickeln Sie analog zur Rechenvorschrift `append` aus Vorlesungsteil 4 eine Haskell-Rechenvorschrift `inject` mit der Signatur `inject :: Bunch m => (Term, Term, Term) -> Pred m`. Für Listen `a`, `b`, `c` soll die Relation `inject (a,b,c)` gelten, wenn `c` diejenige Liste ist, die sich daraus ergibt, dass `b` in der Mitte von `a` eingefügt wird. Ist die Länge von `a` ungerade, so soll `b` nach dem mittleren Element von `a` eingefügt werden; ist die Länge von `a` gerade, so soll `b` in der

Mitte von von `a` eingefügt werden. Die folgenden Aufrufe illustrieren das gewünschte Verhalten:

```
?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]}]
```

- Entwickeln Sie analog zur Rechenvorschrift `good` aus Vorlesungsteil 4 eine Haskell-Rechenvorschrift `bad` zur Erkennung und Generierung "schlechter" Sequenzen von Nullen und Einsen. Schlechte Sequenzen sind dabei durch folgende Regeln definiert:

  1. Die Sequenz [1] ist schlecht.
  2. Wenn s_1 und s_2 zwei schlechte Sequenzen sind, dann ist auch die Sequenz $s1 + +s_2 + +[0]$ schlecht.
  3. Außer den nach Regel 1 und 2 gebildeten Sequenzen gibt es keine weiteren schlechten Sequenzen.

  Intuitiv gilt, dass sich schlechte Sequenzen aus Durchläufen von Binärbäumen in Postfix-Ordnung ergeben, wobei Verzweigungen mit Nullen, Blätter mit Einsen benannt sind.

  Ähnlich wie für `good` sollen Aufrufe für `bad` in folgender Form möglich sein:

```
?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]},...
```