2. Aufgabenblatt zu Funktionale Programmierung vom 04.11.2003, fällig: 11.11.2003

Themen: *Funktionen auf ganzen Zahlen, Paaren und Zeichenreihen, Rekursion*

Für dieses Aufgabenblatt sollen Sie Haskell-Rechenvorschriften zur Lösung der unten angegebenen Aufgabenstellungen entwickeln und für die Abgabe in einer Datei namens `Aufgabe2.hs` ablegen. Kommentieren Sie Ihre Programme in jedem Fall wieder aussagekräftig und benutzen Sie, wo sinnvoll, Hilfsfunktionen und Konstanten.

- Schreiben Sie eine Haskell-Rechenvorschrift `isPowOfTwo`, die für positive ganze Zahlen einschließlich 0 überprüft, ob eine solche ihr als Argument übergebene Zahl eine Potenz von 2 ist.

- Schreiben Sie eine Haskell-Rechenvorschrift `doubleReverse` der Funktionalität "String to String", die eine Eingabezeichenreihe auf eine Zeichenreihe doppelter Länge abbildet, deren erste Hälfte mit der Eingabezeichenreihe und deren zweite Hälfte mit der umgekehrten Eingabezeichenreihe übereinstimmt (z.B.: `doubleReverse` angewendet auf die Zeichenreihe *abcd* liefert die Zeichenreihe *abcddcba*). Schreiben Sie dazu eine geeignete rekursive Hilfsfunktion `reverse`, die eine Eingabezeichenreihe in umgekehrter Folge wieder ausgibt.

- Sei $\Sigma = \{a, b, c, \ldots, z, A, B, C, \ldots, Z\}$ der Zeichenvorrat der Klein- und Großbuchstaben. Eine Zeichenreihe $s = (s_1 s_2 \ldots s_n)$ aus Σ^* heißt *Palindrom*, wenn s vorwärts- und rückwärtsgelesen dieselbe Zeichenreihe darstellt, d.h. wenn gilt

  $$(s_1 s_2 \ldots s_n) = (s_n \ldots s_2 s_1)$$

  Ein klassisches Beispiel für ein Palindrom ist die Zeichenreihe OTTO, nicht aber die Zeichenreihe Otto.

  Schreiben Sie eine rekursive Haskell-Rechenvorschrift `isPal`, die testet, ob eine vorgelegte Zeichenreihe aus Σ^* ein Palindrom ist. Dabei dürfen höchstens Zeichenreihen der Länge 1 verglichen werden und die Funktion `reverse` (auch unter anderem Namen) nicht verwendet werden. *Hinweis*: Σ^* bezeichnet die Menge aller Zeichenreihen über dem Alphabet Σ einschließlich der leeren Zeichenreihe ε.

- In dieser Aufgabe betrachten wir noch einmal die Fibonacci-Zahlen von Aufgabenblatt 1. Offenbar gilt, dass jede Fibonacci-Zahl f_i, $i \geq 2$, die Summe ihrer vorausgehenden zwei Fibonacci-Zahlen ist, d.h. $f_i = f_{i-2} + f_{i-1}$ für $i \geq 2$. Mit $f_0 = 0$ und $f_1 = 1$ gilt damit insbesondere:

  $$0+1=1,\ 1+1=2,\ 1+2=3,\ 2+3=5,\ 3+5=8,\ 5+8=13,\ \ldots$$

  Diese Beobachtung gibt Anlass, die Folge

  $$(0,1),\ (1,1),\ (1,2),\ (2,3),\ (3,5),\ (5,8),\ \ldots$$

  als Folge der *Fibonacci-Paare* $fp_1,\ fp_2,\ fp_3,\ \ldots$ einzuführen mit $fp_1 = (0,1),\ fp_2 = (1,1),\ fp_3 = (1,2)$, usw.

  Mit Hilfe der Fibonacci-Paare wollen wir nun eine Haskell-Rechenvorschrift `fibFix` zur Berechnung der Fibonacci-Zahlen schreiben. Dabei soll sich die Funktion `fibFix` direkt oder indirekt auf zwei Haskell-Rechenvorschriften `nextFp` und `fp` abstützen. Die Funktion `nextFp` bildet dazu ein als Argument übergebenes Fibonacci-Paar fp_n auf das folgende Fibonacci-Paar fp_{n+1} ab, die Funktion `fp` liefert zu einer als Argument übergebenen natürlichen Zahl n das n-te Fibonacci-Paar fp_n. Die Funktion `fibFix` selbst liefert angewendet auf eine natürliche Zahl einschließlich der 0 die entsprechende Fibonacci-Zahl.

  (Ohne Abgabe: Vergleichen Sie die Laufzeiten der Funktion `fibFix` mit derjenigen Ihrer Lösung von Aufgabenblatt 1 zur Berechnung von Fibonacci-Zahlen. Haben Sie eine Erklärung für Ihre Beobachtung?)