Praktische Informatik 3 WS 08/09

Christoph Lüth
Dominik Dietrich
Marcus Ermler
Klaus Hartke
Christian Maeder
Dominik Luecke
Ewaryst Schulz

1. Übungsblatt

Ausgabe: 03.11.08

Bearbeitungszeit: Zwei Wochen

1 *Wurzelbehandlungen* *5 Punkte*

Auf Grund eines Fehlers im Herstellungsprozess ist in Ihrem neuen Laptop die eingebaute Quadratwurzelberechnung defekt! Daher müssen wir ein numerisches Verfahren zur Berechnung der Quadratwurzel einer Zahl implementieren. Ein uraltes Verfahren, welches schon die alten Griechen nutzten, ist nach seinem Erfinder Heron von Alexandrien als Heron-Verfahren (oder Newton-Verfahren[1]) bekannt.

Die Quadratwurzelberechnung nach Heron ist für eine Zahl $a \in \mathbb{R}$ gegeben durch eine Folge $(h_n)_{n\in\mathbb{N}}$, definiert als

$$h_{n+1} = \frac{1}{2}(h_n + \frac{a}{h_n})$$

Dies Folge konvergiert gegen die Quadratwurzel von a:

$$\lim_{n\to\infty} h_n = \sqrt{a}$$

Wir sagen (etwas ungenau), dass die Quadratwurzel aus a bis auf die Genauigkeit ϵ jenes h_n mit dem kleinsten n ist, so dass $|h_n - h_{n+1}| \leq \epsilon$ (d.h. jenes Element der Folge, ab dem der Abstand zwischen den Folgengliedern unter ϵ sinkt).

1. Implementieren Sie eine Funktion

   ```
   wurzel :: Double-> Double-> Double
   ```

   wobei `wurzel a eps` die Quadratwurzel aus `a` mit der Genauigkeit `eps` ist. Als Startwert wählen wir $h_0 = \frac{1+a}{2}$.

   Achten Sie auf eine geeignete Fehlerbehandlung. Ist diese Funktion für alle Eingabewerte definiert?

2. Jetzt können wir zwar die Eingabewerte berechnen, aber wissen nicht, wieviele Iterationen wir dafür benötigen. Implementieren Sie eine Funktion

   ```
   count :: Double-> Double-> Int
   ```

[1] Der britische Computerpionnier Isaac Newton verallgemeinerte dieses Verfahren, um Nullstellen beliebiger reellwertiger Funktionen zu berechnen.

welche nicht p_n, sondern n zurückgibt (d.h. die Anzahl der Iterationen bis zum Abbruch). Wie schnell konvergiert die Folge?

2 *Geometrie, jetzt oder nie.* *5 Punkte*

In dieser Aufgabe betrachten wir einfache geometrische Figuren wie Kreise, Rechtecke, Dreiecke, berechnen ihre Fläche und verschieben sie.

Der grundlegende Begriff ist hierbei ein Punkt $(x, y) \in \mathbb{R}^2$ in der Ebene, gegeben durch zwei reelle Zahlen x, y. Ein Rechteck wird durch zwei Eckpunkte, ein Dreieck durch drei Eckpunkte modelliert; ein Kreis hat natürlich keine Ecken, sondern einen Mittelpunkt und einen Radius.

Für einen Kreis mit dem Radius r ist die Fläche $A = \pi r^2$, für ein Rechteck mit den Kantenlängen a, b ist die Fläche $A = ab$, und für ein Dreieck mit den Kantenlängen a, b, c ist die Fläche $A = \sqrt{s(s-a)(s-b)(s-c)}$, wobei $s = \frac{a+b+c}{2}$.[2] Um aus den Eckpunkten die Kantenlängen zu berechnen, benötigen wir den Abstand Δ zwischen zwei Punkten (p_x, p_y) und (q_x, q_y), welcher sich berechnen läßt als $\Delta = \sqrt{(p_x - q_x)^2 + (p_y - q_y)^2}$.

Fassen wir einen Punkt als Vektor vom Ursprung auf, können wir eine Figur entlang dieses Vektors verschieben, indem wir einfach den Vektor zu den Eckpunkten bzw. dem Mittelpunkt verschieben.

1. Implementieren Sie einen Datentyp

   `Shape`

   welcher Dreiecke, Kreise und Rechtecke darstellt. Der grundlegende Datentyp ist dabei `Point`, welcher einen Punkt in der Ebene repräsentiert. Ein `Point` ist einfach ein Tupel aus zwei `Double`.

2. Implementieren Sie zwei Funktionen

   ```
   area :: Shape-> Double
   move :: Shape-> Point-> Shape
   ```

   welche die Fläche der geometrischen Figur berechnet, und die Figur um den gebenen Punktvektor verschiebt.

Hinweise: Zwei Hilfsfunktionen

```
distance :: Point-> Point-> Double
add :: Point-> Point-> Point
```

[2] Diese Formel geht übrigens auch auf Heron von Alexandrien zurück!

welche den Abstand zwischen zwei Punkten berechnen sowie zwei Punkte addieren (den ersten um den zweiten verschieben) sind sehr hilfreich.

3 *111001101010101* *10 Punkte*

Jetzt kommt es ganz schlimm: bei Ihrem neuen Rechner sind auch noch die Bits kaputt! Deshalb müssen wir auch diese in Haskell nachimplementieren.

Zuerst benötigen wir einen Datentyp `Bit`, der aus zwei disjunkten Werte besteht, mit zwei Funktionen

```
dec1 :: Bit-> Int
enc1 :: Int-> Bit
```

die ein einzelnes Bit in eine ganze Zahl $i \in \{0, 1\}$ verwandeln.

Ein Bitliste ist eine Sequenz einzelner Bits, und ist entweder leer, oder besteht aus einem Bit, vorne an eine Bitliste angehängt.

Implementieren Sie den Datentyp `BitList`, und drei Funktionen

```
encode  :: Int-> BitList
decode  :: BitList-> Int
display :: BitList-> String
```

welche eine ganze Zahl in eine Bitliste kodieren, eine Bitliste in eine ganze Zahl dekodieren, sowie eine Bitliste als Binärzahl anzeigen. Beispiel:

```
*> display (encode 1238923)
"10010111001111000l011"
```