Universität Bremen

Praktische Informatik 3 WS 2023/24

Thomas Barkowsky
Tede von Knorre
Yaroslav Purgin
Tim Sperling

6. Übungsblatt

Ausgabe: 27.11.2023
Abgabe: 04.12.2023

In diesem Übungsblatt wollen wir ganzrationale Funktionen der Form $f(x) = a_n x^n + a_{n-1}x^{n-1} + ... + a_2 x^2 + a_1 x + a_0$ mit $n \in \mathbb{N}_0$, $a_0, a_1, ..., a_n \in \mathbb{R}$ und $a_n \neq 0$ verarbeiten. Für die Polynome der Funktionsgleichungen verwenden wir den folgenden algebraischen Datentyp:

```
data Polynomial = Degree0 Float | DegreeN Float Polynomial
```

Hierbei wird der Konstruktor Degree0 für den Koeffizienten a_0 verwendet, und mit dem Konstruktor DegreeN werden rekursiv die Koeffizienten a_1, a_2, etc. repräsentiert.

6.1 *Funktionen und ihre Werte* — *4 Punkte*

Zunächst wäre eine Funktion

```
makePolynomial :: [Float] → Polynomial
```

nützlich, die Polynome aus einer übergebenen Liste von Koeffizienten erzeugt:

```
makePolynomial [0.7, -3.1, 1.4, 2.1] ⇝ DegreeN 0.7 (DegreeN (-3.1) (DegreeN 1.4 (Degree0 2.1)))
```

Das letzte Element der übergebenen Liste steht also für den Koeffizienten a_0, das erste Element für den n-ten Koeffizienten. Das erzeugte Polynom steht somit für die Funktion $f(x) = 0.7x^3 - 3.1x^2 + 1.4x + 2.1$.

Wird makePolynomial mit der leeren Liste aufgerufen, so soll das Polynom generiert werden, das der Funktion $f(x) = 0$ entspricht.

Als nächstes definieren wir uns eine Funktion degreeOf, die den *Grad eines Polynoms* ermittelt. Der Grad eines Polynoms ist der größte vorkommende Exponent:

```
degreeOf $ makePolynomial [0.7, -3.1, 1.4, 2.1] ⇝ 3
```

Für die Normalparabel $f(x) = x^2$ sieht das dann so aus:

```
degreeOf $ makePolynomial [1, 0, 0] ⇝ 2
```

Was würde uns eine mathematische Funktionsbeschreibung nützen, wenn wir nicht in der Lage wären, Funktionswerte zu berechnen? Wir brauchen also die Haskell-Funktion

```
functionValue :: Polynomial → Float → Float ,
```

die für ein gegebenes Polynom den Funktionswert an der übergebenen Stelle berechnet:

```
functionValue (makePolynomial [0.7, -3.1, 1.4, 2.1]) 3.5 ⇝ -0.9624996
```

Mithilfe von functionValue können wir die Funktion valueTable schreiben, die komplette Wertetabellen erstellt:

```
valueTable :: Polynomial → Float → Float → Float → String
```

Die übergebenen Parameter sind (in dieser Reihenfolge): das Polynom, der kleinste und der größte x-Wert und die Schrittweite für die Werteberechnungen. Die Funktion erzeugt einen String, den wir wieder mit putStrLn ausgeben können.

Beispiel:

(siehe folgende Seite)

```
putStrLn $ valueTable (makePolynomial [0.7, -3.1, 1.4, 2.1]) (-1) 4 0.5 ⇝
     x |    f( x )
-------+-----------
  -1.0 |      -3.1
  -0.5 |     0.537
   0.0 |       2.1
   0.5 |     2.113
   1.0 |       1.1
   1.5 |    -0.413
   2.0 |      -1.9
   2.5 |    -2.837
   3.0 |      -2.7
   3.5 |    -0.962
   4.0 |       2.9
```

Die Tabelle soll insgesamt 19 Zeichen breit sein und aus zwei gleichbreiten Spalten bestehen. In der Mitte der Tabelle befindet sich ein senkrechter Strich ('Pipe'-Symbol) mit je einem Leerzeichen rechts und links davon. Die Zahlen sollen rechtsbündig formatiert sein; Zahlen mit mehr als drei Nachkommastellen sollen auf drei Nachkommastellen gerundet werden. Sonstige Formatierung (Kopfzeile und waagerechter Trennstrich): siehe Unit-Test.

6.2 *Kurvendiskussion* *6 Punkte*

Im zweiten Teil dieses Übungsblatts untersuchen wir die Eigenschaften von Polynomen etwas genauer. Zunächst sind wir an den Nullstellen der Funktionen interessiert. Die Funktion

```
zeros :: Polynomial → [Float]
```

berechnet die Nullstellen von Polynomen ersten Grades (Geraden) und zweiten Grades (Parabeln, Stichwort: p-q-Formel). Für andere Funktionen soll `zeros` die Fehlermeldung `"Polynomial must be of degree 1 or 2."` zurückgeben (Funktion `error`).

Die Nullstellen werden in einer Liste (`[Float]`) zurückgegeben, da für die untersuchten Polynome zwei, eine oder gar keine Nullstelle möglich sind. Wenn es zwei Nullstellen gibt, sollen diese in der Liste nach Größe sortiert sein. Wenn bei einer Parabel zwei Nullstellen in einem Punkt zusammenfallen, soll diese nur einmal in der Liste enthalten sein.

Beispiele:

```
zeros $ makePolynomial [2, 3, -4] ⇝ [-2.350781,0.8507811]
zeros $ makePolynomial [0.6, -1.7] ⇝ [2.8333333]
zeros $ makePolynomial [2,0,2] ⇝ []
zeros $ makePolynomial [0.7, -3.1, 1.4, 2.1] ⇝ *** Exception: Polynomial must be of degree 1 or 2.
```

Nullstellen sind ja die Schnittpunkte des Funktionsgraphen mit der x-Achse. Den Funktionswert, mit dem der Graph die y-Achse schneidet wird durch die Funktion

```
yIntercept :: Polynomial → Float
```

ermittelt:

```
yIntercept (makePolynomial [0.7, -3.1, 1.4, 2.1]) ⇝ 2.1
```

Wenn wir nun wissen wollen, an welchen Stellen unsere Polynome lokale Extrema oder Wendepunkte haben, müssen wir in der Lage sein, die *Ableitung einer Funktion* zu berechnen (die Ableitung einer Funktion beschreibt die Steigung der ursprünglichen Funktion in einem gegebenen Punkt). Die Haskell-Funktion

```
derivativeOf :: Polynomial → Polynomial
```

soll genau das leisten:

```
derivativeOf (makePolynomial [0.7, -3.1, 1.4, 2.1]) ⇝ DegreeN 2.1 (DegreeN (-6.2) (Degree0 1.4))
```

Die Nullstellen der ersten Ableitung einer Funktion liefern uns die lokalen Extrema (oder ggf. auch Wendepunkte) der Funktion:

```
zeros $ derivativeOf (makePolynomial [0.7, -3.1, 1.4, 2.1]) ⇝ [0.24636471,2.706016]
```

Unsere Funktion zeros liefert uns die Nullstellen von Polynomen ersten und zweiten Grades. Um auch Nullstellen von Polynomen höheren Grades berechnen zu können, gibt es das Newton-Verfahren (ein iteratives Näherungsverfahren, siehe https://de.wikipedia.org/wiki/Newtonverfahren). Leider konvergiert das Newton-Verfahren nicht immer und es ist erforderlich, mit einem Startwert in der Nähe der zu ermittelnden Nullstelle zu beginnen. Wir schreiben uns also die Funktion

```
newtonZero :: Polynomial → Float → Float ,
```

die neben dem zu untersuchenden Polynom einen Startwert für das Verfahren erhält. Die Iteration soll abbrechen, wenn sich eine berechnete Nullstelle um weniger als 0.001 von dem im vorhergehenden Iterationsschritt berechneten Wert unterscheidet.

Beispiel: In der oben abgedruckten Wertetabelle können wir sehen, dass unsere Funktion dritten Grades eine Nullstelle in der Nähe von -0.5 haben muss. Wir können also die Iteration mit diesem Wert starten und ermitteln eine Nullstelle bei rund -0.5988:

```
newtonZero (makePolynomial [0.7, -3.1, 1.4, 2.1]) (-0.5) ⇝ -0.59877354
```