2. Übungsaufgabe zu

Fortgeschrittene funktionale Programmierung

Thema: Ströme, Generatoren, Filter, Transformatoren, Memo-Tafeln und Algorithmenmuster

ausgegeben: Di, 21.03.2017, fällig: Di, 28.03.2017

Für dieses Aufgabenblatt sollen Sie Haskell-Rechenvorschriften zur Lösung der im folgenden angegebenen Aufgabenstellungen entwickeln und für die Abgabe in einer Datei namens `AufgabeFFP2.hs` auf **oberstem Niveau in Ihrem Gruppenverzeichnis** ablegen. Kommentieren Sie Ihre Programme aussagekräftig und benutzen Sie, wo sinnvoll, Hilfsfunktionen und Konstanten.

- Von besonderer Bedeutung für die Praxis numerischer Mathematik sind Fehlerabschätzungen für Näherungsverfahren.

  Für die näherungsweise Berechnung von Nullstellen einer stetigen reellen Funktion $f : IR \to IR$ mit einem Vorzeichenwechsel im Intervall $I = [a, b] \subseteq IR$ gemäß des Verfahrens von Aufgabenblatt 1 gilt für jedes Intervall der Folge berechneter Intervalle $I_t = [a_t, b_t]$, $t = 0, 1, 2, \ldots$, mit $I_0 = [a_0, b_0] = [a, b] = I$ die a-priori Fehlerabschätzung, dass die gesuchte Nullstelle z im Intervall $I_t = [a_t, b_t]$ vom Mittelpunkt x_t dieses Intervalls höchstens den Abstand

  $$|x_t - z| \leq \frac{1}{2}(b_t - a_t)$$

  hat. Aufgrund der fortgesetzten Halbierung der Intervalle kann diese Abschätzung unmittelbar auch abhängig von den initialen Intervallgrenzen a_0 und b_0 in folgender Form angegeben werden:

  $$|x_t - z| \leq \frac{1}{2^{t+1}}(b_0 - a_0)$$

  Benutzen Sie diese Beziehung, um auf zwei Arten den Strom der a-priori Fehlerabschätzungen für gegebene initiale Intervallgrenzen a und b zu berechnen:

  ```
  type Low    = Double
  type High   = Double
  type Approx = [Double]
  maxDeviation1 :: Low -> High -> Approx
  maxDeviation2 :: Low -> High -> Approx
  ```

  Bei der Implementierung dürfen Sie davon ausgehen, dass `maxDeviation1` und `maxDeviation2` nur für Argumente aufgerufen werden, wo der `Low`-Wert echt kleiner als der `High`-Wert ist.

  `maxDeviation1` und `maxDeviation2` sollen sich dabei auf unterschiedliche Implementierungen der Funktion 2^t, $t \geq 0$, abstützen, nämlich `powDAC` und `powMemo`.

  `maxDeviation1` stützt sich auf die Funktion höherer Ordnung `divideAndConquer` und deren Hilfsfunktionen `indiv`, `solve`, `divide` und `combine` ab, wobei `divide` und `combine` die Beziehung $2^t = 2^{t-1} + 2^{t-1}$, $t \geq 1$, ausnutzen (vgl. Chapter 3.1, Algorithm patterns, divide-and-conquer):

```
powDAC :: Integer -> Integer
powDAC = divideAndConquer...
```

`maxDeviation2` stützt sich auf

```
powMemo :: Integer -> Integer
powMemo 0 = ...
powMemo t = ...
...
```

ab, wobei `powMemo` analog zum Beispiel zur Berechnung der Fibonacci-Zahlen aus der Vorlesung die Memoizationsidee dieses Beispiels aufgreift und die Berechnung auf eine Memo-Tafel abstützt (vgl. Chapter 2.3, Memoization).

Zusatzaufgabe ohne Abgabe: Vergleichen Sie die Laufzeitverhalten von `maxDeviation1` und `maxDeviation2` und zusätzlich auch anhand von nach und nach größer gewählten Argumenten die von `powDAC` und `powMemo`.

- Jeder Wurf ein Treffer!

  Geworfen wird auf eine Dart-Scheibe mit k verschiedenen Punktwerten. Abschnitte oder Ringe mit doppeltem oder dreifachem Punktwert gibt es nicht, auch nicht das Bull's Eye in der Mitte. Jeder Abschnitt kann in einer Wurffolge jedoch mehrfach getroffen werden.

  Ist es möglich, mit einer Folge von Würfen exakt m Punkte zu erzielen? Ist es möglich, mit genau n Würfen exakt m Punkte zu erreichen? Wieviele Würfe sind mindestens erforderlich, um exakt m Punkte zu erreichen? Stets gilt: kein Wurf geht fehl; jeder Wurf trifft!

  Um diese Fragen zu beantworten, sollen Generatoren, Filter, Transformatoren und Selektoren geschrieben und geeignet kombiniert werden:

```
type Points      = Int      -- Punktwert einer Dart-Scheibe; echt
                            -- positive Zahl
type Dartboard   = [Points] -- Dart-Scheibe charakterisiert durch
                               Liste echt aufsteigender Punktwerte
type Turn        = [Points] -- Punktwerte einer Wurffolge; nur
                            -- Punktwerte, die auf der Scheibe vorkommen
                            -- (d.h., im Dartboard-Wert vorkommen) sind
                            -- moeglich, auch mehrfach moeglich.
type Turns       = [Turn]   -- Strom von Wurffolgen
type TargetScore = Int      -- Gewuenschte Zielpunktsumme > 0
type Throws      = Int      -- Anzahl von Wuerfen einer Wurffolge > 0

gen_turns         :: Dartboard -> Turns
filter_turns_ts   :: Turns -> TargetScore -> Turns
filter_turns_th   :: Turns -> Throws -> Turns
select_turns_minl :: Turns -> Turns
transf_sort_turns :: Turns -> Turns
```

Dabei haben die Funktionen folgende Bedeutung:

- `gen_turns`: Erzeugt einen Strom von Wurffolgen in Übereinstimmung mit den auf der Dart-Scheibe vorhandenen Punkten, z.B. alle Wurffolgen der Länge 1, anschließend alle Wurffolgen der Länge 2, etc. Achten Sie in jedem Fall darauf, dass ihr Generator fair ist, d.h. jede Wurffolge endlicher Länge schließlich liefert und keine Duplikate von Wurffolgen in Form von Permutationen erzeugt (Wurffolgen wie [7,23,12], [23,7,12], [12,7,23], etc. werden als Duplikate voneinander angesehen; nur eine davon soll erzeugt werden).
- `filter_turns_ts`: Filtert aus dem Eingabestrom diejenigen Wurffolgen für die Ausgabeliste heraus, deren aufsummierte Punktewerte die angegebene Zielpunktezahl ergeben.
- `filter_turns_th`: Filtert aus dem Eingabestrom diejenigen Wurffolgen für die Ausgabeliste heraus, die aus der angegebenen Zahl von Würfen bestehen.
- `select_turns_minl`: Wählt aus der Eingabeliste für die Ergebnisliste die Wurffolgen kürzester Länge aus.
- `transf_sort_turns`: Sortiert jede Wurffolge des Eingabestroms für den Ausgabestrom aufsteigend.

Schreiben Sie mithilfe obiger Generatoren, Filter, Transformatoren und Selektoren Funktionen

```
dart_ts   :: Dartboard -> TargetScore -> Turns
dart_tst  :: Dartboard -> TargetScore -> Throws -> Turns
dart_tsml :: Dartboard -> TargetScore -> Turns
```

mit folgender Bedeutung:

- `dart_ts` liefert die (endlich vielen) Wurffolgen mit dem angegebenen Zielpunktewert.
- `dart_tst` liefert die (endlich vielen) Wurffolgen, die den angegebenen Zielpunktewert mit der angegebenen Zahl von Würfen erreichen.
- `dart_tsml` liefert die (endlich vielen) Wurffolgen, die den angegebenen Zielpunktewert mit geringster Wurfzahl ergeben.

Die Reihenfolge der Wurffolgen in den Resultatlisten der Funktionen `dart_ts`, `dart_tst` und `dart_tsml` spielt keine Rolle, allerdings soll jede Wurffolge in einer Resultatliste aufsteigend geordnet sein. Je nach Wahl der Argumente kann die Ergebnisliste jeder Funktion auch leer sein, wenn es keine passenden Wurffolgen gibt.

Aufrufbeispiel:

```
db = [6,7,16,17,26,27,36,37,46,47]
dart_ts db 23    ->> [[6,17],[7,16]]
dart_tst db 55 4 ->> [[7,16,16,16],[6,16,16,17],[6,6,7,36],[6,6,6,37],...]
dart_tsml db 100 ->> [[6,47,47],[7,46,47],[16,37,47],[17,36,47],[17,37,46],...]
dart_ts db 15    ->> []
```