Eine Sache lernt man, indem man sie macht.
Cesare Pavese (1908-1950)
italien. Schriftsteller

Für das Können gibt es nur einen Beweis, das Tun.
Marie von Ebner-Eschenbach (1830-1916)
österr. Schriftstellerin

7. Aufgabenblatt zu Funktionale Programmierung von Mo, 25.11.2019.
Erstabgabe: Mo, 02.12.2019 (12:00 Uhr)

Themen: *Programmieren mit Strömen, Generatoren, Filtern und Selektoren, kombinatorische Probleme*

Besprechung von Aufgabenlösungen, Zweitabgabefrist

- **Teil A, programmiertechnische Aufgaben:** Am ersten Übungsgruppentermin, der auf die *Zweitabgabe* der programmiertechnischen Aufgaben folgt.
- Teil B, Papier- und Bleistiftaufgaben: Entfällt auf den Angaben 5 bis 7.
- **Teil C, Terminhinweise:** Für Übungsgruppen, Vorlesung, Tutorensprechstunde.
- **Zweitabgabefrist:** Siehe „Hinweise zu Organisation und Ablauf der Übung“ auf der Webseite der Lehrveranstaltung.

A Programmiertechnische Aufgaben (beurteilt, max. 100 Punkte)

Schreiben Sie zur Lösung der folgenden Aufgaben ein gewöhnliches Haskell-Skript und legen Sie es in einer (Abgabe-) Datei namens `Angabe7.hs` in Ihrem Home-Verzeichnis auf der Maschine `g0` ab. Der Name der Abgabedatei ist für Erst- und Zweitabgabe ident.

Kommentieren Sie Ihr Programm wieder zweckmäßig, aussagekräftig und problemangemessen. Benutzen Sie, wo sinnvoll, Hilfsfunktionen und Konstanten. Versehen Sie alle Funktionen, die Sie zur Lösung der Aufgaben benötigen, mit ihren Typdeklarationen, d.h. geben Sie deren syntaktische Signatur oder kurz, Signatur, explizit an.

A.1 *Generator-Prinzip:* Schreiben Sie einen Haskell-Generator `generiere_fak_strom`, der den Strom der Fakultätszahlen liefert:

```
generiere_fak_strom :: [Integer]
generiere_fak_strom = ...
```

Aufrufbeispiele:

```
generiere_fak_strom ->> [1,1,2,6,24,120,720,5040,40320..
take 5 generiere_fak_strom ->> [1,1,2,6,24]
filter (> 100) generiere_fak_strom ->> [120,720,5040,40320..
filter (< 10000) generiere_fak_strom ->> [1,1,2,6,24,720,5040..
```

Zusatzaufgabe (*ohne Abgabe*): Warum terminiert der Aufruf
`filter (< 10000) generiere_fak_strom` nicht?

A.2 *Generator/Selektor-Prinzip:* Der Funktionswert der reellen Exponentionalfunktion $exp : \mathbb{R} \to \mathbb{R}$ an der Stelle x wird durch die (endlichen) Partialsummen der Reihe

$$exp\ x = 1 + \frac{x}{1!} + \frac{x^2}{2!} + \frac{x^3}{3!} + \ldots + \frac{x^n}{n!} + \ldots = \sum_{k=0}^{\infty} \frac{x^k}{k!}$$

approximiert.

Schreiben Sie eine Haskell-Rechenvorschrift `approximiere_exp` als Kombination aus einem Generator und einem Selektor, die den Wert der Exponentialfunktion an der Stelle x über die Partialsummen der obigen Reihe bis zu einer gewissen Genauigkeit approximiert. Angewendet auf eine Stelle x und eine vorgegebene Genauigkeit $\epsilon > 0$ liefert `approximiere_exp` diejenige Partialsumme der Reihe, die sich von ihrer Vorgängerpartialsumme betragsmäßig zum ersten Mal höchstens um ϵ unterscheidet:

```
type IR_plus     = Double   -- Nur Werte echt groesser als null
type Stelle      = Double
type Genauigkeit = IR_plus
type Approx_Wert = Double
type Strom       = [Double]

approximiere_exp :: Stelle -> Genauigkeit -> Approx_Wert
approximiere_exp x epsilon = selektiere epsilon (generiere_exp_strom x)

generiere_exp_strom :: Stelle -> Strom
selektiere :: Genauigkeit -> Strom -> Approx_Wert
```

Das bedeutet, der Wert von `generiere_exp_strom x` soll der Strom `[1,1 +` $\frac{x}{1!}$`,1 +` $\frac{x}{1!}$ `+` $\frac{x^2}{2!}$, ... sein; der Wert von z.B. `exp_approx 1.0 0.75` der approximative Wert

$$2.5 = \sum_{k=0}^{2} \frac{x^k}{k!}.$$

Aufrufbeispiele:

```
generiere_exp_strom 1.0  ->> [1.0,2.0,2.5,2.666..67,..
approximiere_exp 1.0 0.5 ->> selektiere 0.5 (generiere_exp_strom 1.0) ->> 2.5
approximiere_exp 1.0 0.2 ->> selektiere 0.2 (generiere_exp_strom 1.0) ->> 2.6..67
```

A.3 *Generator/Filter-Prinzip:* Schreiben Sie einen

(a) Generator `generiere_woerter`, der den Strom endlicher Wörter über dem dreielementigen Alphabet {'a','b','c'} erzeugt. Lexikographisch kleinere Wörter sollen dabei vor lexikographisch größeren Wörtern im Strom aufscheinen.

```
type Woerterstrom = [String]

generiere_woerter :: Woerterstrom
generiere_woerter = ...
```

(b) Filter `filtere_palindrome`, der aus einem Strom von Wörtern diejenigen herausfiltert, die ein Palindrom sind.

```
filtere_palindrome :: Woerterstrom -> Woerterstrom
filtere_palindrome = ...
```

Schreiben Sie Generator und Filter so, dass `filtere_palindrome` auf `generiere_woerter` angewendet werden kann.

Aufrufbeispiele:

```
generiere_woerter ->> ["","a","b","c","aa","ab","ac","ba","bb","bc",
                       "ca","cb","cc","aaa","aab","aac","aba","abb",
                       "abc","aca","acb",..
filtere_palindrome generiere_woerter ->> ["","a","b","c","aa","bb",
                                          "cc","aaa","aba","aca",..
[ s | s <- filtere_palindrome generiere_woerter, length s == 2 ]
  ->> ["aa","bb","cc",..
```

A.4 Zwei Wörter s und t bilden eine *Leiterstufe*, wenn s durch eine von drei Transformationen in t überführt werden kann:

(a) Hinzufügen eines neuen Zeichens am Ende von s.

(b) Löschen des ersten Zeichens von s.

(c) Ersetzen eines beliebigen Zeichens von s durch ein anderes Zeichen.

Bilden s und t eine Leiterstufe, so heißt die Leiterstufe *aufsteigend*, wenn t lexikographisch größer als s ist, sonst *absteigend*. Eine Folge von Wörtern $w_1 - w_2 - w_3 - \ldots - w_k$ heißt *aufsteigende Wortleiter* der Länge k, wenn für alle $1 \leq i < k$ gilt: w_i und w_{i+1} bilden eine aufsteigende Leiterstufe.

Beispiel: Die Wörter `"do"` und `"dog"` bilden eine aufsteigende Leiterstufe, ebenso die Wörter `"dog"` und `"dot"`; die Wortfolge `"do"` – `"dog"` – `"dot"` bildet eine aufsteigende Wortleiter der Länge 3.

Gegeben ist nun ein Wörterbuch mit endlich vielen Worteinträgen über dem Alphabet der Kleinbuchstaben $\{a, b, c, ..., z\}$. Die Einträge im Wörterbuch können Duplikate enthalten und brauchen nicht sortiert zu sein. Gesucht ist die längste aufsteigende Wortleiter, die sich mit den Einträgen aus dem Wörterbuch bilden lässt.

Schreiben Sie eine Haskell-Rechenvorschrift `gib_max_aufsteigende_wortleiter`, die diese Aufgabe erledigt. Gibt es mehr als eine aufsteigende Wortleiter größter Länge, so soll die lexikographisch kleinste dieser Wortleitern geliefert werden. Eine Wortleiter wl heißt *lexikographisch kleiner* als eine Wortleiter wl', wenn wl gleich wl' ist oder wenn das erste Wort von wl, in dem sich wl und wl' unterscheiden, lexikographisch kleiner als das entsprechende Wort von wl' ist (beachte: Die Relation lexikographisch kleiner für Wortleitern ist reflexiv definiert. Genauer, aber weniger schön, müsste die Relation deshalb 'lexikographisch höchstens so groß wie' heißen).

Schreiben Sie zusätzlich zwei Wahrheitswertfunktionen `ist_aufsteigende_leiterstufe` und `ist_aufsteigende_wortleiter`. Die Funktion `ist_aufsteigende_leiterstufe` überprüft, ob zwei beliebige Wörter eine aufsteigende Leiterstufe bilden, die Funktion `ist_aufsteigende_wortleiter`, ob eine Folge beliebiger Wörter eine aufsteigende Wortleiter bilden.

```
type Wort        = String
type Woerterbuch = [Wort] -- nur Werte endliche Laenge; keine Stroeme.
type Wortleiter  = [Wort]
```

```
ist_aufsteigende_leiterstufe    :: Wort -> Wort -> Bool
ist_aufsteigende_wortleiter     :: [Wort] -> Bool
gib_max_aufsteigende_wortleiter :: Woerterbuch -> Wortleiter
```

Aufrufbeispiele:

```
woerterbuch = ["awake","awaken","cat","dig","dog","fig","fin",
               "fine","fog","log","rake","wake","wine"]
gib_max_aufsteigende_wortleiter woerterbuch
  ->> ["dig","fig","fin","fine","wine"] -- Leiterlaenge 5
length (gib_max_aufsteigende_wortleiter woerterbuch) ->> 5
ist_aufsteigende_leiterstufe "fig" "fin" ->> True
ist_aufsteigende_leiterstufe "fin" "fig" ->> False
ist_aufsteigende_wortleiter woerterbuch ->> False
ist_aufsteigende_wortleiter ["dig","fig","fin"] ->> True
ist_aufsteigende_wortleiter ["fin","fig","dig"] ->> False
```

Beachte: Zu kurz im obigen Beispiel wären die aufsteigenden Wortleitern `["dig", "dog","fog",log]` und `["dig","fig","fog","log"]`, beide mit Leiterlänge 4.

Testen Sie alle Funktionen auch mit weiteren eigenen Testdaten (ohne Abgabe).

Wichtig:

1. **Wiederverwendung früherer Lösungsteile:** Wenn Sie einzelne Rechenvorschriften aus früheren Lösungen wieder verwenden möchten, kopieren Sie diese bitte in die neue Abgabedatei ein. Ein `import` (eigener Module) schlägt für die Auswertung durch das Abgabeskript fehl, weil Ihre alte Lösung, aus der importiert wird, nicht mit abgesammelt wird. Deshalb: Kopieren statt importieren zur Wiederwendung!
2. **Abgaben auf `g0` auf gewünschtes Verhalten prüfen:** Aufgabenlösungen werden stets auf der Maschine `g0` unter Hugs überprüft. Überzeugen Sie sich deshalb bitte, dass Ihre Lösungen für dieses und auch alle weiteren Aufgabenblätter sich auf der `g0` unter Hugs wie von Ihnen erwartet und gewünscht verhalten; überzeugen Sie sich bei jeder Abgabe davon. Das gilt besonders, wenn Sie für die Entwicklung Ihrer Haskell-Programme mit einem anderen Werkzeug oder einer anderen Maschine als der `g0` arbeiten!

B Papier- und Bleistiftaufgaben

Entfällt auf den Angaben 5 bis 7.

C Terminhinweise

1. **Übungsgruppen**: Die nächsten Übungsgruppentreffen finden in dieser Woche statt, von heute, Montag, 25.11.2019, bis Donnerstag, 28.11.2019.

2. **Vorlesung:** Der nächste Vorlesungstermin ist morgen, Dienstag, den 26.11.2019, von 08:15 Uhr bis 09:45 Uhr im Informatik-Hörsaal in der Treitlstraße.

3. **Tutorensprechstunde:** Die nächste Tutorensprechstunde findet am Freitag dieser Woche, den 29.11.2019, im `complang`-Labor statt. Sie finden das `complang`-Labor im Innenhof des Institutsgebäudes in der Argentinierstraße 8.

Alle weiteren geplanten Vorlesungs- und Übungsgruppentermine finden Sie auf der Webseite zur Lehrveranstaltung. Mögliche Änderungen werden dort oder/und in der Vorlesung bzw. in den von möglichen Änderungen betroffenen Übungsgruppen bekanntgegeben.