Praktikum: **Grundlagen der Programmierung**
Wintersemester 2008/2009
Prof. Dr. Dr. h.c. M. Broy
F. Deißenböck, Dr. H. Ehler, M. Feilkas, F. Forster, F. Hölzl, B. Hummel,
D. Koss, Dr. M. Kuhrmann, C. Neumann, Dr. M. Spichkova
24.11.2008

Technische Universität München
Fakultät für Informatik

Blatt 5: Funktionale Programmierung II

Aufgabe 1: Schnellere Berechnung der Fibonacci-Zahlen

Auf dem letzten Blatt hatten wir die Fibonacci-Folge betrachtet. Die n-te Fibonacci-Zahl kann man beispielsweise mit folgendem FJava-Programm berechnen:

```
int fib (int n) {
  if (n <= 1)
    return n;
  else
    return fib (n-1) + fib (n-2);
}
```

Hinweise für Tutoren: In dieser Aufgabe soll gezeigt werden, wie man durch eine andere Formulierung des Algorithmus eine deutlich höhere Effizienz erzielt. Ziel ist jedoch lediglich ein Intuitives Verständnis.

a) Schreiben Sie eine Funktion *fibSeq*, die für ein n die Liste der ersten n Fibonacci-Zahlen in umgekehrter Reihenfolge zurückliefert. Der Aufruf mit $n = 6$ sollte also $[8, 5, 3, 2, 1, 1, 0]$ zurückliefern. Die Funktion soll dabei *nicht* die Funktion fib benutzen, sondern stattdessen mit nur einem einzigen rekursiven Aufruf auskommen.

Antwort:

```
Seq<Integer> fibList (int a) {
  if (a == 0) return cons (0);
  else if (a == 1) return cons (1, 0);
  else {
    final Seq<Integer> s = fibList (a-1);
    return concat (cons (first (s) + first(rest(s))), s);
  }
}
```

b) Zeichnen Sie den Aufrufbaum für den Aufruf von fibList(4) und testen Sie die Geschwindigkeit der Funktion für große Werte (40, 60, 100). Vergleichen Sie die Geschwindigkeit und den Aufrufbaum mit dem von der Funktion *fib()*.

Antwort: Der Aufrufbaum ist nur ein Pfad (hier nicht gezeigt). Die Geschwindigkeit von fibList sollte deutlich höher sein als die von fib; für große Werte von n sollte fast keine Verzögerung bemerkbar sein.

Grund hierfür (nicht Teil der Aufgabe!): Die Funktion fib benötigt exponentiell viele Funktionsaufrufe, während fibList nur linear viele benötigt. Die eingesetzte Technik zur Beschleunigung ist unter dem Begriff *dynamische Programmierung* bekannt.

Aufgabe 2: Flugplanungssystem II

Wir betrachten nochmal das Flugplanungssystem von Blatt 3, für das die folgenden Flugdaten gegeben waren:

Von	*Nach*
Berlin	Rom
Berlin	London
Berlin	Paris
Rom	London
London	Madrid
London	Athen
Paris	London
Paris	Madrid
Athen	Berlin
Madrid	Rom

Das Ziel bei dieser Aufgabe besteht darin, für einen gegebenen Start- und Zielort eine Route zu finden, bei der man möglichst selten umsteigen muss.

Hinweise für Tutoren: Der erste Teil der Aufgabe (die Repräsentation) sollte gemeinsam durchgeführt werden, damit alle von ähnlichen Voraussetzungen ausgehen. Der Rest kann dann wieder getrennt in Gruppen bearbeitet werden. Wie weit Sie die Aufgabe wie im Lösungsvorschlag vorstrukturieren, können Sie von Ihrer Gruppe abhängig machen.

a) Wie kann man den oben gegebenen Flugplan in FJava geschickt darstellen? Schreiben Sie eine Funktion *defaultFlights()*, die diesen Plan zurückliefert.

Antwort: Wie stellen die Flüge als eine Sequenz von Strings dar, wobei immer aufeinanderfolgende Paare in der Liste eine (gerichtete) Verbindung darstellen. Alternative Lösungen wäre eine Sequenz in der die Verbindungen wieder als Sequenz stehen.

```
1  // The list of flights used
2  Seq<String> defaultFlights () {
3     return cons (
4        "Berlin", "Rom",
5        "Berlin", "London",
6        "Berlin", "Paris",
7        "Rom", "London",
8        "London", "Madrid",
9        "London", "Athen",
10       "Paris", "London",
```

```
11        "Paris", "Madrid",
12        "Athen", "Berlin",
13        "Madrid", "Rom");
14 }
```

b) Schreiben Sie die Funktion, die für zwei gegebene Orte die Route mit den wenigsten Zwischenstops zurückgibt. Beachten Sie dabei auch den Fall, dass es keine Verbindung gibt oder ein unbekannter Ort eingegeben wird. Testen Sie hierfür die Verbindungen von Rom nach Paris, von Rom nach Rom und von Rom nach Bern (das ja nicht im Flugplan ist).

Antwort: Wir beginnen mit einer Funktion, die für einen gegebenen Ort alle möglichen Folgeorte angibt:

```
1  // Returns all possible next destinations for a location
2  Seq<String> nextStops (String location, Seq<String> flights) {
3     if (isEmpty(flights) || isEmpty(rest(flights)))
4        return emptySeq ();
5     else {
6        final Seq<String> remainder = nextStops (location, rest (rest (flights)));
7        if (location.equals (first(flights)))
8           return concat (cons (first (rest (flights))), remainder);
9        else
10          return remainder;
11    }
12 }
```

Als nächstes konstruieren wir die Funktion *prolong*, die eine Sequez von gegebenen Routen entsprechend den Flugplan verlängert.

```
1  // prolongs all given routes by one step according to the flight plan
2  Seq<Seq<String>> prolong (Seq<Seq<String>> routes, Seq<String> flights) {
3     if (isEmpty(routes))
4        return emptySeq();
5     else
6        return concat (prolong2 (first(routes),
7                                 nextStops (last(first(routes)), flights)),
8                       prolong (rest (routes), flights));
9  }
10
11 // returns the last element of a sequence
12 <T> T last (Seq<T> s) {
13    ensure (!isEmpty (s));
14    if (isEmpty (rest (s)))
15       return first (s);
16    else return last (rest (s));
17 }
18
19 // prolong the given route by all possibilities given in tails
20 Seq<Seq<String>> prolong2 (Seq<String> route, Seq<String> tails) {
21    if (isEmpty (tails))
22       return emptySeq();
23    else
24       return concat (cons (concat (route, cons(first (tails)))),
25                      prolong2 (route, rest (tails)));
26 }
```

Abschließend können wir die Funktion *bestRoute* schreiben. Hierbei werden ausgehend von der einfachsten Route, die nur aus der Startposition besteht, längere Routen konstruiert, bis es entweder keine Routen mehr gibt oder eine Route mit der Zielposition als letztem Element gefunden wird. Damit wir nicht unendlich lange im Kreis laufen können, werden Routen mit doppelten Einträgen aus der Liste der aktuellen Routen entfernt.

```
1  // returns shortest route (returns emptySeq() if no route exists)
2  Seq<String> bestRoute (String from, String to, Seq<String> flights) {
3     return bestRoute2 (cons(cons(from)), to, flights);
4  }
5
6  // returns shortest route (returns emptySeq() if no route exists)
7  Seq<String> bestRoute2 (Seq<Seq<String>> routes, String to, Seq<String> flights) {
8     final Seq<Seq<String>> toRoutes = filterLast(routes, to);
9
10    if (isEmpty (routes))
11       return emptySeq ();
12    if (!isEmpty(toRoutes))
13       return first (toRoutes);
14    else
15       return bestRoute2 (filterRep (prolong (routes, flights)), to, flights);
16 }
17
18 // returns those sequences, where t is the last element
19 <T> Seq<Seq<T>> filterLast (Seq<Seq<T>> s, T t) {
20    if (isEmpty (s))
21       return emptySeq ();
22    else if (last (first (s)).equals (t))
23       return concat (cons (first (s)), filterLast (rest (s), t));
24    else return filterLast (rest (s), t);
25 }
26
27 // returns those sequences, where the last element appears more than once
28 <T> Seq<Seq<T>> filterRep (Seq<Seq<T>> s) {
29    if (isEmpty (s))
30       return emptySeq ();
31    else if (count (last (first(s)), first(s)) > 1)
32       return filterRep (rest (s));
33    else return concat (cons (first (s)), filterRep (rest (s)));
34 }
35
36 // returns how often t appears in s
37 <T> int count (T t, Seq<T> s) {
38    if (isEmpty (s))
39       return 0;
40    else if (t.equals (first (s)))
41       return 1 + count (t, rest (s));
42    else return count (t, rest (s));
43 }
```

Aufgabe 3: Permutationen

Die Permutation einer Sequenz ist eine Sequenz die die gleichen Elemente enthält, aber möglicherweise in einer anderen Reihenfolge. Beispielsweise wäre für $[1, 2, 5]$ sowohl $[2, 5, 1]$ aber auch $[1, 2, 5]$ eine Permutation. Zu einer gegebenen Sequenz der länge n gibt es $n!$ Permutationen.

Schreiben Sie eine Funktion, die für eine gegebene Sequenz s die Sequenz aller Permutationen von s ausgibt. Gehen Sie dabei davon aus, dass kein Element in s doppelt vorkommt.

Hinweise für Tutoren: Zu dieser Aufgabe gibt es absichtlich fast keine Hilfestellung in der Angabe. Lassen Sie in der Übung alle Studenten alleine oder in kleinen Gruppen an diesem Problem arbeiten, wobei Sie entsprechend der Stärke der Studenten Hilfestellung geben müssen. Erinnern Sie vor Bearbeitung der Aufgabe nochmal an das empfohlene Vorgehen:

- Aufgabenstellung verstehen
- Wie sieht die Signatur der Funktion aus?
- Überlegen: Wie würde man das Problem „von Hand“ lösen?
- Wie kommt die Rekursion ins Spiel?
- Wichtig: Systematisches Vorgehen hilft hier mehr als *trial & error*!

Antwort:

```
1  // Create all permutations of a sequence
2  <T> Seq<Seq<T>> perm (Seq<T> s) {
3      if (isEmpty(s))
4          return cons((Seq<T>)emptySeq());
5      else
6          return pickAndPerm (emptySeq(), s);
7  }
8
9  // Helper function for picking an element from t and adding it to the front of all
10 // permutations of the remaining elements
11 <T> Seq<Seq<T>> pickAndPerm (Seq<T> s, Seq<T> t) {
12     if (isEmpty (t))
13         return emptySeq ();
14     else
15         return concat (appendToAll(first(t), perm(concat(s, rest(t)))),
16                        pickAndPerm (concat(s, cons(first(t))), rest(t)));
17 }
18
19 // Appends t to the front of all sequences in s
20 <T> Seq<Seq<T>> appendToAll(T t, Seq<Seq<T>> s) {
21     if (isEmpty(s))
22         return emptySeq ();
23     else
24         return concat (cons (concat (cons(t), first(s))), appendToAll (t, rest(s)));
25 }
```

Eine etwas „geschicktere“ und kürzere Variante sieht wie folgt aus. Allerdings stellt man fest, dass die Verständlichkeit durch die Kompaktheit etwas leidet.

```
1  // Create all permutations of a sequence
2  <T> Seq<Seq<T>> perm (Seq<T> s) {
3      return permHelp(emptySeq(), emptySeq(), s);
4  }
5
```

```
6  // Helper function for perm
7  <T> Seq<Seq<T>> permHelp(Seq<T> t, Seq<T> l, Seq<T> r) {
8     if (isEmpty(r))
9        return emptySeq();
10    else if (isEmpty(rest(r)) && isEmpty(l))
11       return cons(concat(t, cons(first(r))));
12    else
13       return concat(
14          permHelp(concat(t, cons(first(r))), emptySeq(), concat(l, rest(r))),
15          permHelp(t, concat(l, cons(first(r))), rest(r))
16       );
17 }
```