Name: _____ Vorname: _____
Matrikelnummer:_____ Testat: _____

Paradigmen der Programmierung
Praktikum 2

Hinweise für die Informationssuche:

1. Schauen Sie ins Skript und in die Vorlesungsfolien
2. Ziehen Sie die API-Dokumentation zu Rate: http://www.scala-lang.org/api/current/index.html
3. **Fragen Sie während des Praktikums**
4. Verwenden Sie die Dateien `Main.scala` und `Lst.scala` von **www.gm.fh-koeln.de/ehses/paradigmen**

Aufgabe 1. (Listenklasse) In der funktionalen Programmierung sind Listen die grundlegende Datenstruktur. Daher sollen Sie hier einmal eine eigene Listenimplementierung schreiben.

Dazu ist anzumerken:

- Die Implementierung der Methoden erfolgt in einer Klasse. Die Liste ist also immer der this-Parameter,
- Die Methoden entsprechen den Methoden der Klasse List. Die Implementierung ist ziemlich verschieden.
- Orientieren Sie sich auch an den vorhandenen Funktionen!
- Die Buchstaben A und B bezeichnen Typparameter. Die Angabe ist nur nötig, wenn sonst der Typ nicht klar wäre: z.B. Aufruf der Methode `Lst.empty[A]` oder `Lst.empty[B]`.
- [B >: A] bedeutet, B ist ein Obertyp von A.

Sie sollen die folgenden Methoden der Klasse **Lst[A]** implementieren:

- **`foldLeft[B] (z: B) (f: (B, A) => B): B`**
  Gegenstück zu `foldRight`: fasst linksassoziativ alle Elemente, beginnend mit z, unter der Operation f zusammen („Addition“).
- **`map[B](f: A => B): Lst[B]`**
  Die Ergebnisliste enthält die Funktionsresultate von f auf den Elemente der Ausgangsliste.
- **`flatMap[B](f: A => Lst[B]): Lst[B]`**
  Wie map, nur, dass f Listen liefert, die mit append aneinandergehängt werden.
- **`filter(p: A => Boolean): Lst[A]`**
  Die Ergebnisliste enthält die Elemente der Liste, für die p erfüllt ist.
- **`exists(p: A => Boolean): Boolean`**
  Es gibt in der Liste ein Element, das p erfüllt
- **`apply(n: Int): A`**
  Das n-te Element der Liste (`NoSuchElementException` bei leerer Liste)

Gehen Sie bei der Implementierung experimentell vor. D.h. verwenden Sie bei der Implementierung unterschiedliche Methoden:

- Prozedurale Implementierung mit While-Schleife.
- Rekursive Implementierung (muss nicht endrekursiv sein)
- Verwendung der Funktion `foldRight`. (z.B. `map`, `flatMap`, `filter`, `exists`)
- Implementieren Sie `map` oder `filter` mit `flatMap`.

Aufgabe 2. Testen Sie Ihre Lösungen mit dem Objekt `Main`. Finden Sie heraus, was die For-Schleifen bedeuten und was sie mit der Klasse `Lst` zu tun haben (Hinweis: Eclipse kann Ihnen evtl. helfen, wenn Sie den Cursor über die Elemente der For-Anweisung bewegen. Oder schauen Sie einfach, was passiert, wenn Sie einzelne Methoden auskommentieren (map, flatMap, filter). Ignorieren Sie die Warnung (`withFilter`...) oder fragen Sie einfach nach.

Augabe 3. (Experiment mit unvollständigen Datenstrukturen)

Kopieren Sie die Datei `Lst.scala` in eine Datei `Streams.scala` und ändern Sie alle Vorkommen von `Lst` in `Streams` um (bis hierher haben Sie nur einen anders benannte Listenklasse). Jetzt sollen Sie nur 2 kleine Änderungen machen:

In **`object`** `Streams` ändern Sie die Methode `cons`:

```
def cons[A](hd: A, tl: => Streams[A]): Streams[A] = {
  new Streams[A] {
    lazy val cell = Some((hd, tl))
  }
}
```

In **`object`** `Streams` fügen Sie die Methode `from` hinzu:

```
/* alle Zahlen ab n */
def from(n: Int): Streams[Int] = cons(n, from(n + 1))
```

Wie Sie sich denken können, gibt es viele Ähnlichkeiten zwischen `Streams` und `Lst`. Der Unterschied besteht darin, dass `Streams` erst bei Bedarf wirklich konstruiert werden. Das erklärt auch, warum die scheinbar unendliche Rekursion bei `from` (s.o.) funktioniert.

Solange Sie nicht wirklich nach der Unendlichkeit nachfragen, können Sie beliebig große Datenstrukturen definieren. z.B. den `Streams` aller Primzahlen (dabei helfen `from` und `filter`).

Hinweis: man kann beliebige Algorithmen formulieren. Vor der Ausgabe, sollten Sie aber dann `takeAsList(n)` vornehmen (um die Ausgabe auf n-Elemente zu beschränken und eine bequeme Ausgabe per `toString` zu nutzen).

Anmerkungen:

Anstelle von `Lst` kennt die Scala-Bibliothek die Klasse `List`

Anstelle von `Streams` kennt Scala die Klasse `Stream`

(beide sind natürlich erheblich perfekter programmiert)