Probeklausur

Programmieren II für Wirtschaftsinformatiker

Universität:	Technische Universität Berlin
Kurs/Modul:	Programmieren II für Wirtschaftsinformatiker
Bearbeitungszeit:	180 Minuten
Erstellungsdatum:	September 6, 2025

Bearbeitungszeit: 180 Minuten.

Aufgabe 1.

(a) Erläutern Sie kurz die Unterschiede zwischen funktionaler Programmierung und objektorientierter Programmierung. Diskutieren Sie dabei insbesondere die Behandlung von Seiteneffekten und Zustandsänderungen.

(b) In Scala sollen Listenoperationen demonstriert werden. Geben Sie ein kurzes Beispiel mit map und flatMap an, das folgende Transformationen zeigt:

- Verdoppeln der Werte einer Liste von Ganzzahlen
- Erzeugen einer Liste, die zu jedem Element die doppelte Zahl und deren Quadrat enthält

(c) Beschreiben Sie grob, wie Nebenläufigkeit in Scala über Futures realisiert wird und welche typischen Fallstricke dabei auftreten können.

(d) Geben Sie eine kurze pseudocode- oder Scala-ähnliche Skizze an, wie eine Tail-Recursive-Funktion zur Berechnung der n-ten Fibonacci-Zahl aussieht. Erläutern Sie, warum Tail-Recursion wichtig ist.

Aufgabe 2.

(a) Gegeben sei der Case Class-Baustein in Scala:

```
case class Transaction(id: String, amount: Double, status: String)
```

Schreiben Sie eine funktionale Funktion, die eine Liste von Transaktionen filtern soll, sodass nur Transaktionen mit Betrag $>$ 1000 und Status "COMPLETED" übrig bleiben. Verwenden Sie keine var-Variablen.

(b) Introduceieren Sie eine einfache Pattern-Matching-Behandlung zur Auswertung der folgenden Datenstruktur:

```
sealed trait Result
case class Success(value: String) extends Result
case class Failure(code: Int, reason: String) extends Result
```

Schreiben Sie eine Funktion, die bei `Success` den Wert zurückgibt und bei `Failure` eine Fehlermeldung erzeugt.

(c) Erklären Sie den Unterschied zwischen Immutable und Mutable Collections in Scala und geben Sie ein kurzes Beispiel, bei dem eine immutable Collection durch Kopie erweitert wird.

(d) Beschreiben Sie kurz das Konzept von Traits in Scala und nennen Sie zwei Typ-Fälle, in denen Traits sinnvoll eingesetzt werden.

Aufgabe 3.

(a) Skizzieren Sie eine einfache Architektur eines scala-basierten Dienstes zur Verarbeitung von Bestell-Events, der mit Akka-Actors arbeitet. Nennen Sie mindestens drei Arten von Nachrichten (Messages) und eine grobe Supervising-Strategie.

(b) Erklären Sie, wie Akka-Actors sicher Nebenläufigkeit ermöglichen, und nennen Sie einen typischen Nachteil gegenüber reinen Futures.

(c) Gegeben seien zwei einfache Akka-Actors:

```
class Printer extends Actor {
  def receive = {
    case s: String => println(s)
  }
}
class Master extends Actor {
  val p = context.actorOf(Props[Printer], "printer")
  def receive = {
    case msg: String => p ! msg
  }
}
```

Beschreiben Sie grob, wie eine Nachricht von einem Sender an den Master weitergeleitet wird und wie der Printer-Actor die Nachricht ausgibt.

(d) Diskutieren Sie kurz die Vor- und Nachteile der Verwendung von Akka-Streams für datenintensive BI-Aufgaben im Vergleich zu klassischen Sammlungs-basierten Ansätzen.

Aufgabe 4.

(a) Entwerfen Sie in wenigen Absätzen eine grobe Specifikation für einen Scala-basierten ETL-Dienst, der Daten aus einer Quelle (z. B. CSV-Datei) in ein Zielsystem überführt. Legen Sie die Hauptfunktionen fest (Input, Transformation, Output) und benennen Sie passende Typen.

(b) Formulieren Sie zwei messbare Lernziele aus dem Kurskontext, die sich direkt auf die praktische Umsetzung des ETL-Dienstes beziehen.

(c) Nehmen Sie an, dass Sie eine einfache Fehlertoleranz implementieren möchten. Skizzieren Sie eine Mechanik, wie fehlgeschlagene Transformationsschritte protokolliert und später erneut versucht werden können.

(d) Reflektieren Sie in einem Satz, wie die Konzepte der Funktionsprogrammierung bei der Entwicklung dieses Dienstes zu besserer Wartbarkeit beitragen können, insbesondere im Hinblick auf Nebenläufigkeit und Fehlerbehandlung.

Lösungen

Bearbeitungszeit: 180 Minuten.

Aufgabe 1.

(a) Funktionale Programmierung (FP) und objektorientierte Programmierung (OP) unterscheiden sich primär in ihrem Denkstil, der Handhabung von Nebenwirkungen sowie der Art, wie Zustände modelliert werden.

- FP betont Unveränderlichkeit (Immutability) und Funktionsanwendungen ohne Nebeneffekte. Funktionen sind in der Regel referenzäquivalent, d. h. dieselben Eingaben liefern immer dieselben Ausgaben. Seiteneffekte werden minimiert, wodurch reasoning about code erleichtert wird.
- OP modelliert Zustände durch Objekte, die Status über Methoden verändern können. Klassenhierarchien, Vererbung und Polymorphie ermöglichen flexibles Design, können aber zu komplexeren Zustands- und Nebenwirkungspfaden führen.
- In FP (z. B. in Scala) werden oft unveränderliche Datentypen, Monaden (Option, Either, Future) und rein funktionale Muster verwendet, während in OO-Ansätzen oft mutable Zustände, Vererbungshierarchien und side effects vorherrschen.
- Behandlung von Seiteneffekten: FP strebt Planschichten von reinen Funktionen an; Nebenwirkungen werden explizit durch Monaden (z. B. Future, IO) oder durch strukturierte Effekte modelliert. In der OOP kann der Zustand eines Objekts unmittelbare Seiteneffekte in Methodenaufrufen verursachen.

(b) Beispiel in Scala mit map und flatMap:

```
val nums = List(1, 2, 3, 4)

val doubled: List[Int] = nums.map(_ * 2)
// Ergebnis: List(2, 4, 6, 8)

val doubledAndSquare: List[(Int, Int)] =
  nums.flatMap(n => List((n * 2, n * n)))
// Ergebnis: List((2,1), (4,4), (6,9), (8,16))
```

(c) Nebenläufigkeit in Scala über Futures:

- Futures ermöglichen asynchrone Berechnungen, die auf einem ExecutionContext (typischerweise einem Thread-Pool) laufen. Der Resultatzugriff erfolgt via Abstraktionen wie map/flatMap/kombinieren mit for-Comprehensions.
- Typische Fallstricke:
  - Blockierende Aufrufe innerhalb eines Future (z. B. Await.result) können an den vorhandenen Threads blockieren und Deadlocks oder Thread-Teppiche verursachen.
  - Fehlendes Fehler-Handling: Exceptions in Futures müssen through recover/recoverWith oder pattern matching on Failure abgefangen werden.

  - ExecutionContext-Überlastung oder falsches Thread-Pooling (z. B. CPU-bound Tasks auf einem IO-Pool) kann zu schlechter Skalierbarkeit führen.
  - Race conditions bei gemeinsam genutzten Ressourcen außerhalb von Actors oder Futures ohne Synchronisation.

- Typische Best Practices: vermeide blockierende Aufrufe, nutze nicht-reaktive Side-Effects innerhalb Futures, lagere IO-gebundene Aufgaben in spezialisierte ExecutionContexts aus, verwende geeignete Pattern Matching für Failure, kombiniere Futures deterministisch.

(d) Tail-Recursive-Funktion zur n-ten Fibonacci-Zahl (Scala-ähnliche Skizze):

```
import scala.annotation.tailrec

def fib(n: Int): BigInt = {
  @tailrec
  def go(a: BigInt, b: BigInt, k: Int): BigInt =
    if (k == 0) a
    else go(b, a + b, k - 1)

  if (n < 0) throw new IllegalArgumentException("n must be >= 0")
  go(0, 1, n)
}
```

Tail-Recursion ist wichtig, weil der Compiler (bei entsprechend annotiertem tailrec) keine wachsenden Stack-Frames erzeugt, sondern die Rekursion in eine Schleife überführt. Dadurch bleiben Speicherverbrauch und Stack-Verbrauch konstant, was bei großen n von Vorteil ist.

Aufgabe 2.

(a) Gegeben sei der Case Class-Baustein in Scala:

```
case class Transaction(id: String, amount: Double, status: String)
```

Schreiben Sie eine funktionale Funktion, die eine Liste von Transaktionen filtern soll, sodass nur Transaktionen mit Betrag > 1000 und Status C̈OMPLETED¨brig bleiben. Verwenden Sie keine var-Variablen.

```
def filterCompletedLarge(transactions: List[Transaction]): List[Transaction] =
  transactions.filter(t => t.amount > 1000 && t.status == "COMPLETED")
```

(b) Introduceieren Sie eine einfache Pattern-Matching-Behandlung zur Auswertung der folgenden Datenstruktur:

```
sealed trait Result
case class Success(value: String) extends Result
case class Failure(code: Int, reason: String) extends Result
```

Schreiben Sie eine Funktion, die bei `Success` den Wert zurückgibt und bei `Failure` eine Fehlermeldung erzeugt.

```
def evaluateResult(r: Result): String = r match {
  case Success(value) => value
  case Failure(code, reason) => s"Error $code: $reason"
}
```

(c) Erklären Sie den Unterschied zwischen Immutable und Mutable Collections in Scala und geben Sie ein kurzes Beispiel, bei dem eine immutable Collection durch Kopie erweitert wird.

Immutable Collections erzeugen neue Instanzen bei Änderungen; die ursprüngliche Collection bleibt unverändert. Beispiele: List, Vector, Immutable Seq. Mutable Collections erlauben in-place-Änderungen, z. B. scala.collection.mutable.ListBuffer, ArrayBuffer.

```
val a = List(1, 2, 3)
val b = a :+ 4        // neue List -> Vector-ähnlich, ursprüngliche a bleibt unverändert
// a = List(1, 2, 3), b = List(1, 2, 3, 4)

import scala.collection.mutable.ListBuffer
val buf = ListBuffer(1, 2, 3)
buf += 4              // modify in place
// buf = ListBuffer(1, 2, 3, 4)
```

(d) Beschreiben Sie kurz das Konzept von Traits in Scala und nennen Sie zwei Typ-Fälle, in denen Traits sinnvoll eingesetzt werden.

Traits ermöglichen die Mischung von Verhalten in Klassen, ohne eine feste Klassenhierarchie erzwingen zu müssen. Sie können abstrakte Member (Methoden/Val) deklarieren und konkrete Implementierungen liefern. Traits unterstützen Mehrfachvererbung (Mixin-Komposition) und dienen als saubere Schnittstelle bzw. als wiederverwendbare Bausteine.

Zwei sinnvolle Typ-Fälle:

- Mixin von Cross-Cutting-Concerns wie Logging oder Auditing: trait Logger def log(msg: String): Unit = println(msg)
- Bereitstellung standardisierter Schnittstellen oder Hilfen, z. B. serialisierbare Utilities oder Validierungslogik, z. B. trait JsonSerializable def toJson: String

Aufgabe 3.

(a) Skizzieren Sie eine einfache Architektur eines scala-basierten Dienstes zur Verarbeitung von Bestell-Events, der mit Akka-Actors arbeitet. Nennen Sie mindestens drei Arten von Nachrichten (Messages) und eine grobe Supervising-Strategie.

Architektur (stichpunktartig):

- Akka-Actor-System mit Roles: OrderServiceActor, InventoryActor, PaymentActor, NotificationActor.
- Haupt-Nachrichten (Messages):
  - PlaceOrder(orderId: String, items: List[String], customer: String)
  - ConfirmOrder(orderId: String)
  - CancelOrder(orderId: String, reason: String)
  - OrderStatusRequest(orderId: String)
  - OrderStatusResponse(orderId: String, status: String)
- Grobe Supervising-Strategie: One-for-one (bei Fehlerszenarien der Unter-Actors Neustart des fehlerhaften Actors), mit Eskalation bei persistierenden Fehlern; ggf. Restart- oder Resume-Strategien je nach Art des Fehlers.
- Persistenz/Monitoring: ggf. Event-Sourcing oder Logging der Zustandsübergänge; Supervisory-Strategy kann pro Actor angepasst werden.

(b) Erklären Sie, wie Akka-Actors sicher Nebenläufigkeit ermöglichen, und nennen Sie einen typischen Nachteil gegenüber reinen Futures.

- Nachrichten-basierte Kommunikation erzwingt Asynchronität; jeder Actor verarbeitet seine Postfach-Nachrichten seriell, wodurch interne Zustandsänderungen thread-sicher sind.
- Kein geteiltes Mutable-State; Zustandszugriffe erfolgen über Messaging, wodurch Race Conditions minimiert werden.
- Typische Nachteil gegenüber reinen Futures: Höherer Overhead durch Message-Passing-Mechanismen; komplexere Debugbarkeit bei vielen Akteuren; eventuelle Latenz aufgrund von asynchroner Koordination im Vergleich zu direkter Futures-Komposition.

(c) Gegeben seien zwei einfache Akka-Actors:

```
class Printer extends Actor {
  def receive = {
    case s: String => println(s)
  }
}
class Master extends Actor {
  val p = context.actorOf(Props[Printer], "printer")
  def receive = {
    case msg: String => p ! msg
  }
}
```

Beschreiben Sie grob, wie eine Nachricht von einem Sender an den Master weitergeleitet wird und wie der Printer-Actor die Nachricht ausgibt.

Antwort:

- Ein Sender schickt eine String-Nachricht an den Master (z. B. via masterRef ! "Hallo").
- Der Master empfängt die Nachricht in seiner receive-Block, und leitet sie asynchron an den Printer weiter: p ! msg.
- Der Printer empfängt die Nachricht in seinem eigenen receive-Block und führt println aus, wodurch der Text auf der Konsole erscheint.

(d) Diskutieren Sie kurz die Vor- und Nachteile der Verwendung von Akka-Streams für datenintensive BI-Aufgaben im Vergleich zu klassischen Sammlungs-basierten Ansätzen.

Vorteile von Akka-Streams:

- End-to-End-Backpressure und streaming-basierte Verarbeitung großer Datenmengen (Speicherbedarf wird kontrolliert).
- Natürliche Integration in verteilte Architekturen; robuste Fehlerbehandlung durch supervision.
- Asynchrone Verarbeitung mit guter Throughput-Bandbreite.

Nachteile:

- Höhere Komplexität bei Setup, Debugging und Fehlersuche.
- Overhead durch Actor- und Stream-ABstraktionen im Vergleich zu rein sammlungsbasierten Ansätzen.
- Für einfache, kleine BI-Pipelines ist der Mehraufwand oft nicht gerechtfertigt.

Aufgabe 4.

(a) Entwerfen Sie in wenigen Absätzen eine grobe Specifikation für einen Scala-basierten ETL-Dienst, der Daten aus einer Quelle (z. B. CSV-Datei) in ein Zielsystem überführt. Legen Sie die Hauptfunktionen fest (Input, Transformation, Output) und benennen Sie passende Typen.

Spektrum des ETL-Dienstes:

- Input: Lese-Funktion ReadSource(path: String): Seq[RawRow]
- Transformation: Transform(raw: RawRow): Either[String, TransformedRow]
- Output: WriteTarget(rows: Seq[TransformedRow], dest: String): Either[String, Unit]

```
case class RawRow(fields: Map[String, String])
case class TransformedRow(fields: Map[String, String])
```

Beispielhafte Typen:

```
type InputRow  = Map[String, String]
type OutputRow = Map[String, String]
```

Ein einfacher Ablauf könnte so aussehen:

```
def readSource(path: String): Seq[RawRow] = ...

def transform(row: RawRow): Either[String, TransformedRow] = ...

def writeTarget(rows: Seq[TransformedRow], dest: String): Either[String, Unit] = ...
```

(b) Formulieren Sie zwei messbare Lernziele aus dem Kurskontext, die sich direkt auf die praktische Umsetzung des ETL-Dienstes beziehen.

- Die Teilnehmenden können eine scala-basierte ETL-Pipeline entwerfen, implementieren und automatisiert gegen eine Testdatenmenge validieren (z. B. mit ScalaTest) und dabei Reinheit der Transformation sicherstellen (keine Mutable-Variablen, klare Fehlerpfade).
- Die Teilnehmenden können funktionale Konzepte (Immutability, Pattern Matching, Future-basiertes I/O) gezielt in der Pipeline einsetzen, um Nebenläufigkeit und Fehlertoleranz zu demonstrieren (z. B. parallele Transformationen, Backpressure-Handling).

(c) Nehmen Sie an, dass Sie eine einfache Fehlertoleranz implementieren möchten. Skizzieren Sie eine Mechanik, wie fehlgeschlagene Transformationsschritte protokolliert und später erneut versucht werden können.

Vorschlag:

- Fehlgeschlagene Transformationen werden in einer persistierten Fail-Pipeline (Append-only Log) aufgezeichnet mit Feldern wie timestamp, inputRow, errorCode, reason.
- Ein Retry-Mechanismus mit Exponential-Backoff (z. B. nach Fail-Event ein Re-try nach 1s, 2s, 4s, ...) bis max.RetryCount.
- Bei erfolgreichem Retry wird die transformierte Zeile in die Zielpipeline integriert; bei anhaltendem Fehler ist der Eintrag in einem separaten Audit-Store markiert.
- Monitoring-Dashboard: Anzahl der fehlgeschlagenen Zeilen, Retry-Rate, Kingfisher-ähnliche Kennzahlen.
- Optionale kaskadierende Re-Aktivierung durch Policy (z. B. Landmark-Recovery, Exponential-Backoff mit Jitter).

(d) Reflektieren Sie in einem Satz, wie die Konzepte der Funktionsprogrammierung bei der Entwicklung dieses Dienstes zu besserer Wartbarkeit beitragen können, insbesondere im Hinblick auf Nebenläufigkeit und Fehlerbehandlung.

Durch den Fokus auf unveränderliche Datenstrukturen, reine Funktionen und explizite Fehlerpfade wird der Code besser testbar, vorhersehbar und leichter parallelisierbar; Nebenläufigkeit lässt sich modellieren, ohne geteilten Zustand, wodurch Fehlersituationen besser handhabbar werden.