Übungsaufgabe

Grundlagen von Scala: Syntax, Werte- und Referenztypen, Pattern Matching, Sammler.

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

Aufgabe 1: Grundlagen von Scala: Syntax, Werte- und Referenztypen, Pattern Matching, Sammler

In dieser Aufgabe arbeiten Sie mit den Kernkonzepten von Scala. Gehen Sie sorgfältig auf die folgenden Teilaufgaben ein. Ziel ist es, sich mit der Syntax, dem Unterschied zwischen Wert- und Referenztypen, Pattern Matching sowie grundlegenden Sammlern vertraut zu machen.

a) Syntax und Typen Beschreiben Sie, wie sich unveränderliche Werte (val) von veränderlichen Werten (var) unterscheiden und welche Rolle Typinferenz spielt. Skizzieren Sie, wie Funktionen in Scala definiert werden (def) und wie Typen oft implizit inferred werden. Formulieren Sie ein kurzes, typisiertes Beispiel (nur als Beschreibung der Struktur, nicht als Lösung) das val, var und def umfasst.

b) Werte- vs. Referenztypen Geben Sie an, ob folgende Konzepte in Scala typischerweise als Werte- oder Referenztypen betrachtet werden, und begründen Sie kurz Ihre Einschätzung:

- Integer-Werte wie 42
- Zeichenketten (String)
- Arrays
- Listen (List)
- optionale Typen (Option[T])

c) Pattern Matching Beschreiben Sie den Zweck von Pattern Matching in Scala. Skizzieren Sie, wie ein einfaches Muster-Match-Beispiel aufgebaut ist (ohne konkreten Code bereitzustellen): Pattern, Cases, ggf. Guards. Erklären Sie, wann Pattern Matching sinnvoll ist, besonders bei verschachtelten bzw. algebraischen Datentypen.

d) Sammler (Collections) Geben Sie eine kurze Übersicht über grundlegende Sammler-Typen in Scala (z. B. Seq/List, Vector, Set, Map) und deren Eigenschaften bzgl. Unveränderlichkeit. Beschreiben Sie zwei typische Operationen, die auf Sammlern häufig verwendet werden (z. B. Abbildung/Filtern, Aggregation), und nennen Sie die Konzepte von map, filter, reduce/fold, ohne konkrete Implementierung vorzulegen.

Aufgabe 2: Fortgeschrittene Pattern Matching und Sammler in Scala

In dieser Aufgabe vertiefen Sie Pattern Matching mit algebraischen Datentypen (ADTs) und arbeiten mit Sammlern, um einfache Transformations- und Abfrageaufgaben zu erfüllen. Verwenden Sie klare, lesbare Muster und vermeiden Sie nebenläufige Konzepte.

a) ADTs und Pattern Matching Definieren Sie in Worten ein kleines ADT-Design (z. B. einen einfachen Ausdrucksbaum) und erläutern Sie, wie Pattern Matching genutzt wird, um verschiedene Baumknoten zu unterscheiden. Beschreiben Sie zwei konkrete Matches (ohne Implementierung), die unterschiedliche Baumknoten behandeln.

b) Pattern Matching mit Variablen und Guards Erläutern Sie, wie Varianten mit Variablen gebunden werden und Guards (wenn-Bedingungen) genutzt werden können. Geben Sie zwei Beispiele in Worten an, bei denen Guards eine Bedingung auf Werte im Muster prüfen.

c) Sammler-Übungen: Transformationen mit Map/Filter/FlatMap Beschreiben Sie eine Aufgabe, in der Sie eine Liste von Ganzzahlen zuerst filtern, dann transformieren (z. B. quadrieren) und schließlich ein Aggregat (z. B. Summe) berechnen. Geben Sie die Schritte in Form von Teilschritten an (ohne konkrete Programme zu liefern).

d) Komposition von Sammlern und Pattern Matching Skizzieren Sie eine Aufgabe, bei der Sie zunächst eine Sammlung von Produkten nach Kategorie filtern, anschließend per Pattern Matching auf eine Fallunterscheidung reagieren (z. B. unterschiedliche Behandlung je nach Produktkategorie) und schließlich Ergebnisse in einer Map speichern. Beschreiben Sie die Struktur der Lösung in Worten.

Lösungen

Lösung zu Aufgabe 1: Grundlagen von Scala: Syntax, Werte- und Referenztypen, Pattern Matching, Sammler

a) Syntax und Typen

- Unterschied val vs var:
  - val bindet einen unveränderlichen Wert. Der Wert, der hinter der Bindung steht, bleibt nach der Zuweisung fest.
  - var bindet eine veränderliche Referenz. Der zugewiesene Wert kann später verändert werden.
- Typinferenz:
  - Der Compiler versucht, den Typ aus dem rechten Ausdruck abzuleiten. Typen können explizit angegeben werden, sind aber oft nicht notwendig.
  - Wenn der Typ nicht eindeutig ist, muss explizite Typangabe erfolgen.
- Funktionsdefinitionen (def):
  - Funktionen werden deklariert als def name(param1: Type1, param2: Type2): ReturnType = ...
  - Typen der Parameter und des Rückgabewerts können oft weggelassen werden, wenn sie eindeutig inferiert werden.

Beispielstruktur (Beschreibung der Struktur, nicht als konkreter Lösungscode): Eine unveränderliche Bindung (val) mit Typ T1; eine veränderliche Bindung (var) mit Typ T2; eine Funktion (def) mit einer Parameterliste [Parameter1: T1, Parameter2: T2] und Rückgabetyp R, deren Körper eine Transformation von Eingaben auf das Ergebnis R beschreibt. Die Typen können durch den Compiler aus den rechten Ausdrücken abgeleitet oder explizit angegeben werden.

b) Werte- vs. Referenztypen

- Integer-Werte wie 42: Werttypen (Int, also AnyVal). Sie werden oft effizient als Werte behandelt.
- Zeichenketten (String): Referenztypen (meistens AnyRef). Strings in Scala sind unveränderlich, aber das Objekt ist eine Referenz auf eine Speicherstruktur.
- Arrays: Referenztypen. Java-Arrays (und damit auch Scala-Arrays) sind veränderlich, das Array-Objekt selbst ist ein Referenztyp.
- Listen (List): Referenztypen im Sinn der JVM-Objektorientierung; List ist eine unveränderliche Datenstruktur (Persistent/immutable), die als Objekt referenziert wird.
- optionale Typen (Option[T]): Referenztyp (Konstrukte wie Some(...) und None). Option ist ein Container-Objekt, das entweder einen Wert enthält oder nichts enthält.

Begründung: In Scala (und der JVM) werden primitive Typen wie Int, Double etc. als Werttypen (AnyVal) behandelt; Sammlungen und ADTs bestehen typischerweise als Objekte (Referenztypen, meist unveränderliche Strukturen, aber es gibt auch veränderliche Varianten in der Mutable-Sammlungskategorie).

c) Pattern Matching

- Zweck: Pattern Matching erlaubt das strukturierte Entpacken und Unterscheiden von Werten anhand ihrer Form (z. B. Konstruktoren von ADTs) und dessen Innerem.
- Aufbau eines einfachen Muster-Match-Beispiels (ohne konkreten Code):
  - Pattern: beschreibt die Form des erwarteten Ausdrucks (z. B. eine Konstante, eine Addition oder eine komplexere Baumstruktur).
  - Cases: einzelne Zweige, die auf spezifische Pattern-Formen reagieren.
  - Guards (Optional): Bedingungen, die zusätzlich prüfen, ob das Pattern wirklich passend ist (z. B. bestimmte Wertemerkmale).
- Sinnhaftigkeit: Pattern Matching ist besonders nützlich bei verschachtelten bzw. algebraischen Datentypen (ADTs), da es erlaubt, alle möglichen Bauformen explizit zu behandeln und fehlende Fälle sicher abzuwickeln.

d) Sammler (Collections)

- Grundlegende Sammler-Typen (Beispiele, unveränderliche Varianten):
  - Seq/List: sequentiell geordnete Sammler; List ist eine unveränderliche, verkettete Liste.
  - Vector: unveränderliche Sequenz mit gutem Zufallszugriff und guter-like-Performance für viele Operationen.
  - Set: unveränderliche Menge (ohne Duplikate); ungeordnete Sammlung von Elementen.
  - Map: unveränderte Zuordnung von Schlüsseln zu Werten.
- Zwei typische Operationen:
  - Abbildung/Filtern: map transformiert jedes Element, filter wählt Elemente gemäß einem Prädikat aus.
  - Aggregation: reduce bzw. fold fasst Elemente zu einem Endwert zusammen.

Lösung zu Aufgabe 2: Fortgeschrittene Pattern Matching und Sammler in Scala

a) ADTs und Pattern Matching

- ADT-Design: Ein kleiner Ausdrucksbaum (Expr) als algebraischer Datentyp. Typischer Aufbau:
  - Ein Wurzeltyp Expr, der durch mehrere Knotenformen realisiert wird, z. B. Const(n: Int) für Konstanten, Add(l: Expr, r: Expr) für Addition, Mul(l: Expr, r: Expr) für Multiplikation.
  - Zur Abbildung der Baumstruktur kommen weitere Knotenformen in Frage (z. B. Var(name: String), Sub, Div etc.), je nach Anwendungsbedarf.
- Pattern Matching wird genutzt, um verschiedene Baumknoten anhand ihrer Konstruktorform zu unterscheiden (z. B. Const, Add, Mul) und anschließend rekursiv oder iterativ weiterzuarbeiten.
- Zwei konkrete Matches (ohne Implementierung):
  - Ein Match, das einen Const-Knoten behandelt (z. B. Extraktion des zugrundeliegenden Werts).
  - Ein Match, das einen Add-Knoten behandelt (z. B. rekursive Behandlung der linken und rechten Teilbäume und anschließendes Kombinieren der Ergebnisse).

b) Pattern Matching mit Variablen und Guards

- Bindung von Variablen im Muster: Ein Muster kann Variablen binden (z. B. eine Teillösung eines Teilbaums), sodass der gebundene Wert im Case weiter verwendet werden kann.
- Guards (wenn-Bedingungen): Zusätzliche Bedingungen, die erfüllt sein müssen, damit der Case gewählt wird.
- Zwei Beispiele in Worten:
  - Beispiel 1: Wenn der Pattern-Branch einen Const(n) fängt, wird zusätzlich geprüft, ob n größer als 0 ist (Guard). Nur wenn diese Bedingung erfüllt ist, wird der Case ausgewählt.
  - Beispiel 2: Ein Pattern Add(a, b) wird gebunden, und ein Guard prüft, dass beide Unterbäume Konstanten sind oder dass die Summe der Werte eine bestimmte Bedingung erfüllt.

c) Sammler-Übungen: Transformationen mit Map/Filter/FlatMap

- Aufgabe (ohne konkrete Implementierung): Beginne mit einer Liste von Ganzzahlen, filtere zuerst eine Teilmenge (z. B. gerade Zahlen), transformiere dann jedes Element (z. B. Quadriere es) und berechne schließlich ein Aggregat (z. B. die Summe der transformierten Werte).
  - Teilschritt 1: Startliste L von Ganzzahlen.

  - Teilschritt 2: L1 = L.filter(prädikat) // z. B. gerade Zahlen
  - Teilschritt 3: L2 = L1.map(fun) // z. B. Quadriere jedes Element
- Teilschritt 4: Ergebnis = L2.foldLeft(0)($_{+}$)//$z.B. Summeder Werte$

d) Komposition von Sammlern und Pattern Matching

- Aufgabenbeschreibung in Worten: Gegeben ist eine Sammlung von Produkten, aus der zunächst alle Produkte einer bestimmten Kategorie gefiltert werden. Anschließend erfolgt eine Reaktion auf unterschiedliche Produktkategorien mittels Pattern Matching (z. B. unterschiedliche Behandlung je nach Kategorie). Abschließend werden die Ergebnisse in einer Map zusammengeführt, z. B. Kategorie -> aggregiertes Ergebnis.
- Struktur der Lösung in Worten:
  - Schritt 1: Filtere die Produkt-Sammlung nach der gewünschten Kategorie.
  - Schritt 2: Wende Pattern Matching auf jedes gefilterte Produkt an, um je Kategorie eine spezifische Verarbeitung zu realisieren.
  - Schritt 3: Sammle die Resultate pro Kategorie in einer Map (z. B. Map[Kategorie, BerechnungsErgebnis]).