Lernzettel

Funktionale Programmierung: Prinzipien und Anwendungen

Universität:	Technische Universität Berlin
Kurs/Modul:	Softwaretechnik und Programmierparadigmen
Erstellungsdatum:	September 19, 2025

Lernzettel: Funktionale Programmierung: Prinzipien und Anwendungen

(1) Einführung.
Die funktionale Programmierung (FP) betrachtet Funktionen als zentrale Bausteine der Software. Typische Merkmale sind Unveränderlichkeit von Daten, reine Funktionen ohne Nebeneffekte und höhere Ordnung Funktionen. FP betont Abstraktion, Modularisierung und Explizitheit von Berechnungen. In vielen Sprachen wie Haskell, OCaml, F, Scala oder Scheme lässt sich FP principiennah umsetzen.

(2) Zentrale Prinzipien.

Unveränderlichkeit.
Daten werden nach ihrer Erstellung nicht verändert; stattdessen entstehen neue Werte.

Unveränderlichkeit: x wird nach Zuweisung nicht verändert.

Reine Funktionen.
Eine Funktion hängt nur von ihren Eingaben ab und hat keine Nebeneffekte.

Pure Funktion: $f(x)$ liefert immer denselben Output zu x.

Referentielle Transparenz.
Ein Ausdruck kann durch seinen reduzierten Wert ersetzt werden, ohne das Programmverhalten zu ändern.

$$f(x) = y \;\Rightarrow\; \text{ersetze } f(x) \text{ durch } y \text{ überall}$$

Höhere Ordnung Funktionen.
Funktionen können Eingaben von anderen Funktionen akzeptieren oder als Ergebnisse liefern.

$$(f \circ g)(x) = f(g(x)).$$

Funktionskomposition.
Kombination mehrerer Funktionen zu einer neuen Funktion.

Komposition: $(f \circ g)(x) = f(g(x))$.

Abstraktion und Pattern Matching (DTypen).
Algebraische Datentypen wie Sum- und Produkt-Typen ermöglichen klare Strukturen. Pattern Matching erleichtert die De-konstruktion von Werten.
Beispiel (schematisch):

$$\text{data List a} = \text{Nil} \mid \text{Cons } a(Lista)$$

(3) Datenstrukturen und Typen in FP.

Algebraische Datentypen (ADT).
Sum- und Produkt-Typen modellieren Varianten und Kombinationsmöglichkeiten von Daten.

$$\text{Maybe } a = \text{Nothing} \mid \text{Just } a$$

$$\text{Either } ab = \text{Left } a \mid \text{Right } b$$

Pattern Matching.
Falls-Darstellungen ermöglichen, ADTs fallbasiert zu verarbeiten. Beispiel (schematisch):

$$\text{case } z \text{ of Just } x \to \cdots \mid \text{Nothing} \to \ldots$$

(4) Typen und Typinferenz.
Viele FP-Sprachen unterstützen Typsysteme mit Typinferenz. Funktionen haben oft generische Typen, z. B. $\forall a.\ a \to a$ für Identität.

$$\text{Identität: } \mathrm{id}(x) = x \quad \Rightarrow \quad \text{Typ Id} = a \to a$$

(5) Higher-Order- und Funktionsmuster.

Map, Filter, Fold.

$$\text{map } f\ [x_1, \ldots, x_n] = [f(x_1), \ldots, f(x_n)]$$

$$\text{filter } p\ [x_1, \ldots, x_n] = [x_i \mid p(x_i)]$$

$$\text{foldr } g\ z\ [x_1, \ldots, x_n] = g(x_1, g(x_2, \ldots g(x_n, z) \ldots))$$

(6) Nebeneffekte und Abstraktion.
FP zielt darauf ab, Nebeneffekte zu isolieren (z. B. durch Monaden in Sprachen wie Haskell). Dadurch bleiben Berechnungen referentiell transparent, während IO, Logging etc. kontrolliert behandelt werden.

$$\text{Monaden } (M, \text{return}, (>>=)) \text{ kapseln Effekte ab.}$$

(7) Anwendungen und Vorteile.
- Klarheit durch reine Funktionen und Abstraktion.
- Leichte Komposition von Verarbeitungspipelines.
- Einfache Parallelisierung durch Unveränderlichkeit.
- Formalisierung von Algorithmen (z. B. Map-Reduce, Stream-Verarbeitung).

(8) Beispiele in typischen FP-Sprachen.
Beispiel-Pattern (Haskell-ähnlich):

```
map f [1,2,3] - [f 1, f 2, f 3]
foldr (:)  [] [1,2,3] - [1,2,3]
```

(9) Praktische Hinweise zur Lernanwendung.
- Beginne mit einfachen Funktionen und unveränderlichen Strukturen.
- Implementiere kleine Pipelines aus map, filter, fold.
- Nutze Pattern Matching statt komplexer If-Else-Ketten.

(10) Kurzes Übungsbeispiel (Auszug).
Gegeben sei eine Liste von Ganzzahlen. Schreibe eine Funktion, die alle geraden Zahlen quadriert und nur die ersten drei Ergebnisse zurückliefert. Beispielskizze (Pseudocode/Funktional):

$$\text{take } 3(\text{map } (\lambda x \to x^2)(\text{filter } (\lambda x \to x \bmod 2 = 0)\ [1,2,3,4,5,6]))$$