Pseudocode-Baukasten (funktional)

Konventionen in diesem Pseudocode

- Funktionsdefinition: `f(x) = ...`
- Pattern-Matching: `match Wert with | Muster -> Ausdruck`
- Listen: `[]` (leer), `x :: xs` (Kopf x, Rest xs)
- Keine Zuweisungen/Mutation. Ergebnisse entstehen nur über **Rückgabe**.
- Höhere Funktionen: `map(f, xs)`, `filter(p, xs)`, `fold(op, start, xs)`
- Komposition: `(g ∘ f)(x) = g(f(x))`P
- Guards/Bedingungen: `if … then … else …`

---

1) Pure Functions & Immutabilität

1.1 Referentielle Transparenz

```
// pure: gleicher Input ⇒ gleicher Output; keine Nebeneffekte
add(a, b) = a + b
```

Erklärung: `add` hängt ausschließlich vom Eingabesatz ab. Kein Lesen/Schreiben außerhalb, keine Zeitabhängigkeit. Austausch durch den Wert ist immer korrekt (referentielle Transparenz).

Übersetzungs-Hinweise (ohne Code):

- In Python Funktionen so schreiben, dass sie nur Parameter lesen und genau **einen Rückgabewert** liefern.
- Keine globalen Variablen lesen/schreiben.
- Keine in-place Änderungen an übergebenen Strukturen; stattdessen **neue** Strukturen erzeugen.

1.2 Unveränderliche Daten (Listen/Records)

```
// "Erweitern" erzeugt eine neue Liste; Original bleibt unverändert
append(xs, x) = xs ++ [x]

// "Entfernen" erzeugt eine neue Liste ohne p-passende Elemente
remove_if(p, xs) =
  match xs with
  | []      -> []
  | y::ys   -> if p(y) then remove_if(p, ys)
                       else y :: remove_if(p, ys)
```

Erklärung: Beide Funktionen erstellen **neue** Ergebnisse, statt das Eingabe-Objekt zu modifizieren.

Übersetzungs-Hinweise:

- In Python: nie Methoden verwenden, die **in-place** arbeiten; stattdessen Ausdrucksformen wählen, die **Kopien**/neue Objekte liefern.

---

2) Rekursive Muster (anstatt Schleifen)

2.1 Falten/Reduzieren (allgemein)

```
fold(op, acc, xs) =
  match xs with
  | []      -> acc
  | y::ys   -> fold(op, op(acc, y), ys)
```

Erklärung: Universelles Rekursionsschema: ersetzt Schleifen. Viele Aggregationen (Summe, Produkt, Max, …) sind Spezialisierungen.

Beispiele als Spezialisierungen (weiterhin Pseudocode):

```
sum(xs)      = fold( (a, b) -> a + b , 0, xs )
product(xs)  = fold( (a, b) -> a * b , 1, xs )
max_of(xs)   = fold( (m, b) -> if b > m then b else m, -∞, xs )
```

2.2 Map & Filter

```
map(f, xs) =
  match xs with
  | []      -> []
  | y::ys   -> f(y) :: map(f, ys)

filter(p, xs) =
  match xs with
  | []      -> []
  | y::ys   -> if p(y) then y :: filter(p, ys)
                       else      filter(p, ys)
```

Erklärung: Ohne Mutation und Schleifen: `map` transformiert, `filter` selektiert. Beide sind pure.

---

3) Verschachtelte Summen (Nested Structures)

3.1 Summe über beliebig verschachtelte Listen

```
// Typidee: Node := Zahl | Liste[Node]
sum_nested(node) =
  match node with
  | Zahl z   -> z
  | Liste xs -> fold( (acc, e) -> acc + sum_nested(e), 0, xs )
```

Erklärung: Rekursion folgt der Struktur. Zahlen sind **Basisfälle**; Listen werden per `fold` über ihre Elemente reduziert, wobei jedes Element ggf. erneut rekursiv verarbeitet wird.

Übersetzungs-Hinweise:

- In Python: Typprüfung nur nutzen, um „Zahl vs. Liste“ zu unterscheiden; Rekursion beibehalten; keine Zähler/Schleifen.

---

4) Zinseszins (rein funktional)

4.1 Rekursive Periodenverzinsung

```
compound(principal, rate_per_period, periods) =
  if periods == 0 then principal
  else compound(principal * (1 + rate_per_period),
                rate_per_period,
                periods - 1)
```

Erklärung: Jede Periode multipliziert den Betrag, reduziert die Periodenzahl **bis zum Basisfall** 0. Kein Zustand außerhalb, keine Seiteneffekte.

4.2 Als Fold über eine „Periodenliste“

```
// periods_to_list(n) = [1, 2, …, n]  (nur konzeptionell)
compound_fold(principal, r, n) =
  fold( (amount, _) -> amount * (1 + r), principal, periods_to_list(n) )
```

Erklärung: Identisch zur Rekursion, aber als Fold formuliert (zeigt Austauschbarkeit von Rekursion und Fold).

Übersetzungs-Hinweise:

- In Python: keine Schleifen nötig; rekursive Form oder eine Fold-ähnliche Reduktion verwenden. Keine Zwischenmutation.

---

5) „Dateisuche“ ohne IO (rein funktionale Modellierung)

> Dateisystem-IO ist **seiteneffekthaft**. Für funktionale Prüfung modellierst du den Verzeichnisbaum als **reine Datenstruktur** (Baum). Dann bleibt die Suche pure.

5.1 Datentyp & reine Traversierung

```
// Tree := File(name) | Dir(name, children : Liste[Tree])
ends_with_txt(name) = suffix(name, ".txt")  // pure Prädikatfunktion

find_txt(tree) =
  match tree with
  | File(n)        -> if ends_with_txt(n) then [n] else []
  | Dir(_, kids)   -> fold( (acc, t) -> acc ++ find_txt(t), [], kids )
```

Erklärung: Kein echtes Filesystem, sondern ein **Baum** als Eingabe. Ausgabe ist die Liste der Pfade/Dateinamen mit `.txt`. Rekursion + Fold, keine Mutation.

Übersetzungs-Hinweise:

- In Python bei echter IO: reine Kernlogik behalten (Traversal + Prädikat) und IO-Schicht **separat** kapseln. Tests gegen **Baum-Attrappe** (Mock) schreiben.

---

6) Seiteneffekte eliminieren (Refactoring-Muster)

6.1 „Lesen → Rechnen → Schreiben“ entkoppeln

```
// impur: read() + compute() + write() vermischt ⇒ schwer testbar
// purer Kern als reine Funktion:
compute_core(input_data) = … // nur Berechnung, pure

// IO-Ränder getrennt:
program(io_read, io_write) =
  let data     = io_read()
  let result   = compute_core(data)
  in  io_write(result)
```

Erklärung: Funktionaler Kern ist pure; IO passiert nur in dünnen Randfunktionen. So bleibt das Meiste testbar/deterministisch.

6.2 Keine In-Place-Updates: „Kopie mit Änderung“

```
// Record als unveränderlicher Wert
update_record(record, key, new_val) =
  record_copy = clone(record)        // konzeptionell: neue Struktur
  record_copy[key] = new_val         // logisch: ergibt ein NEUES record
  return record_copy
```

Erklärung: Statt Zustand zu mutieren, entsteht ein **neues** Exemplar mit gewünschter Abweichung (strukturelle Persistenz).

Übersetzungs-Hinweise:

- In Python: Wörter wie „kopieren“, „neu erzeugen“ statt „ändern“. z. B. Konstrukte nutzen, die **frische Objekte** liefern.

---

7) Imperativ → Funktional (ohne imperativen Pseudocode zu zeigen)

> Du wirst in der Prüfung oft imperativen Code sehen. Strategie zum **Umdenken** in funktional:

Checkliste (mental, ohne Code):

1. **Seiteneffekte markieren** (liest/schreibt global? in-place? IO?). → In „Rand“ auslagern.
2. **Loop erkennen** (akkumuliert etwas?). → Ist es **map**, **filter** oder **fold**? Wähle das passende Schema.
3. **Zähler/Index** vorhanden? → Entfernen; stattdessen Strukturrekursion (`x :: xs`) oder höheres Muster.
4. **Zwischenergebnisse** in Variablen? → Durch **Rückgaben**/Komposition ersetzen.
5. **Mutationen** an Collections? → Ersetze durch **neue** Collections.
6. **Abbruchbedingung** extrahieren. → Das ist dein **Basisfall**.

---

8) Komposition & kleine Bausteine

8.1 Pipeline-Stil (deklarativ)

```
// von Rohdaten zu Ergebnis über klare Schritte:
normalize = map( λx -> (x − μ) / σ )
positives = filter( λx -> x > 0 )
total     = fold( (a, b) -> a + b, 0 )

pipeline(xs) = (total ∘ positives ∘ normalize)(xs)
```

Erklärung: Keine Zwischenzustände, nur Funktionskomposition. Jeder Schritt ist pure.

Übersetzungs-Hinweise:

- In Python gedanklich in „Schritt-Funktionen“ organisieren und diese hintereinander anwenden; keine Sammelvariable, kein `append`.

---

9) Typische Prüfungsaufgaben – funktionale Lösungsformen

1. **„Summiere nur positive Quadrate einer Liste“**

```
sum_pos_squares(xs) =
  xs
  |> filter( λx -> x > 0 )
  |> map(    λx -> x * x )
  |> fold(   (a,b) -> a + b, 0 )
```

2. **„Flache Summe über nested Struktur (Zahlen/Listen)“** → Siehe **3.1** (`sum_nested`).

3. **„Zinseszins mit jährlichem Zusatzbeitrag c“**

```
compound_with_contrib(P, r, n, c) =
  if n == 0 then P
  else compound_with_contrib(P*(1+r) + c, r, n - 1, c)
```

4. **„Texte mit Suffix filtern, Namen transformieren, sortiert ausgeben“**

```
txt_names_sorted(tree) =
  find_txt(tree)                  // pure Traversierung
  |> map( base_name_without_ext )  // pure Umbenennung
  |> sort_immutable                // pure Sortierfunktion ⇒ neue Liste
```

---

10) Prüfungs-Taktik (funktional denken)

- **Benenne Basisfälle zuerst.** (leere Liste, `n == 0`, Blatt im Baum)
- **Wähle ein Schema:** `map`, `filter`, `fold`, Rekursion, Komposition.
- **Keine Zuweisungen.** Wenn du dich ertappst, eine Variable „weiterzuschreiben“, formuliere es als **neue Rückgabe**.
- **Trenne IO von Logik.** Erst Daten rein, dann pure Berechnung, dann Ausgabe.
- **Teste im Kopf mit kleinen Beispielen.** (1-2 Elemente, Randfälle)
- **Bevorzuge Ausdrucks- statt Anweisungsdenken.** („Was ist der Wert?“ statt „Wie arbeite ich ihn ab?“)