1) Listen rekursiv verarbeiten: *head* & *tail*

```
def summe_vl(liste):
    match liste:
        case []:
            return 0                        // Basisfall: leere Liste → neutrales Element 0
        case [head, *tail]:
            match head:
                case int() | float():
                    return head + summe_vl(tail)           // Kopf ist Zahl → addieren + Rekursion auf Rest
                case list():
                    return summe_vl(head) + summe_vl(tail) // Kopf ist Liste → in Tiefe absteigen + Rest
                case _:
                    return summe_vl(tail)                  // Nicht-numerisch → ignorieren, mit Rest weitermachen
```

Erklärung:

- *head* = erstes Element, *tail* = Liste ohne erstes Element.
- **Basisfall** liefert 0, damit Summen korrekt „von unten“ aufbauen.
- **Strukturrekursion**: Jede Stufe verarbeitet nur den Kopf, den Rest übergibt sie wieder an sich selbst.
- **Keine Mutation**: Wir erzeugen nur Rückgabewerte; Eingaben bleiben unverändert.
- **Pure Funktion**: Gleiches `liste` → gleiches Ergebnis; keine Seiteneffekte.

---

2) Das Faltmuster (*fold*): Schleifenersatz im Funktionalstil

```
def fold(op, akk, liste):
    match liste:
        case []:
            return akk                        // Wenn nichts mehr da ist: Akkumulator als Ergebnis
        case [head, *tail]:
            return fold(op, op(akk, head), tail)  // Verknüpfe Kopf mit Akkumulator, rekursiv weiter
```

Erklärung:

- `fold` ist das **universelle Aggregationsschema**: aus vielen Werten wird **ein** Wert.
- `op` ist eine **pure Verknüpfung** (z. B. „addiere“, „multipliziere“, „nimm Maximum“).
- Kein Zustand „wächst“ heimlich; der **Akkumulator** wird **als Wert** weitergereicht.
- Viele Aufgaben (Summe, Produkt, Max, Zählen, Verkettung) sind nur **Spezialfälle** von `fold`.

Spezialisierungen (konzeptuell):

```
def summe(xs)     = fold( (a,b) -> a + b , 0 , xs )
def produkt(xs)   = fold( (a,b) -> a * b , 1 , xs )
def maximum(xs)   = match xs:
                        case [head, *tail]: fold( (m,x) -> (x if x>m else m), head, tail )
```

---

3) Transformieren & Auswählen: *map* und *filter*

```
def mappe(f, liste):
    match liste:
        case []:
            return []
        case [head, *tail]:
            return [ f(head) ] ++ mappe(f, tail)   // Neue Liste aus transformiertem Kopf + rekursivem Rest
```

```
def filtere(p, liste):
    match liste:
        case []:
            return []
        case [head, *tail]:
            return [head] ++ filtere(p, tail) if p(head)
                   else        filtere(p, tail)    // Nur Elemente behalten, die das Prädikat erfüllen
```

Erklärung:

- **map**: „Wende Funktion **auf jedes Element** an“, ohne zu sagen **wie** iteriert wird.
- **filter**: „Behalte **nur** Elemente, die **Bedingung** erfüllen“.
- Beide sind **pure** und erzeugen **neue** Listen (keine In-Place-Änderungen).
- Zusammensetzbar zu **Pipelines** (siehe unten).

---

4) Verschachtelte Strukturen elegant: *Nested Summe*

```
def summe_nested(node):
    match node:
        case int() | float():
            return node                         // Basisfall: Zahl → direkt deren Wert
        case list():
            return fold( (acc, e) -> acc + summe_nested(e), 0, node ) // Liste → Elemente rekursiv reduzieren
        case _:
            return 0                            // Unerwartetes ignorieren neutral
```

Erklärung:

- **Form folgt Struktur**: Was wie eine Liste von Listen aussieht, wird **rekursiv** behandelt.
- Zahlen sind **Blätter** (direkter Wert), Listen sind **innere Knoten** (rekursiv + `fold`).
- **Neutralität**: Nicht-Numerisches zählt nicht mit → **robuste** Definition.

---

5) Zeitliche Prozesse ohne Zähler: *Zinseszins* als Rekursion/Fold

```
def zinseszins(P, r, n):
    match n:
        case 0:
            return P                                  // Nach 0 Perioden: unverändert
        case _:
            return zinseszins(P * (1 + r), r, n - 1)  // Eine Periode anwenden, Periodenzahl verringern
```

```
def zinseszins_fold(P, r, n):
    return fold( (A, _) -> A * (1 + r), P, [1…n] )   // Gedachte Periodenliste, reine Reduktion
```

Erklärung:

- Jede Periode ist **dieselbe Transformation** → ideal für Rekursion oder `fold`.
- **Basisfall** `n=0` beendet die Rekursion.
- **Kein Zustand** außerhalb: Nur Werte rein, Wert raus.

Mit konstantem Beitrag `c` je Periode:

```
def zinseszins_mit_beitrag(P, r, n, c):
    match n:
        case 0:
            return P
        case _:
            return zinseszins_mit_beitrag(P * (1 + r) + c, r, n - 1, c)
```

Erklärung:

- Reihenfolge pro Periode ist **klar** definiert (erst Zins, dann Beitrag).
- Der **neue Betrag** ist ausschließlich eine **Funktion** des alten Betrags → pure.

---

6) Reine Dateisuche: Baum statt IO

```
// Datenmodell (rein): File(name) | Dir(name, children)
def endet_mit_txt(name):
    return endswith(name.lower(), ".txt")     // reines Prädikat (keine IO)

def finde_txt(knoten):
    match knoten:
        case File(name):
            return [name] if endet_mit_txt(name) else []
        case Dir(_, kinder):
            return fold( (acc, kid) -> acc ++ finde_txt(kid), [], kinder )
```

Erklärung:

- **IO ist Seiteneffekt** → in Prüfungen den **Kern** als **Baum** modellieren (reine Daten).
- Traversierung: **Baumrekursion** + `fold` über Kinder → **alle Ebenen** ohne Schleife.
- Ergebnis ist **nur** von Eingabe abhängig (pure); keine globale Ablage, keine Mutation.

---

7) Unveränderliche Updates: „neu statt ändern“

```
def dict_setze_immut(d, key, val):
    return {**d, key: val}     // konzeptionell: Kopie mit überschriebenem Eintrag

def liste_plus(xs, x):
    return xs ++ [x]           // neue Liste mit angehängtem Element

def entferne_if(p, xs):
    match xs:
        case []:
            return []
        case [head, *tail]:
            return entferne_if(p, tail) if p(head)
                   else [head] ++ entferne_if(p, tail)
```

Erklärung:

- **Immutabilität**: Bestehende Strukturen bleiben **unangetastet**.
- „Änderung“ bedeutet: **neue** Struktur mit gewünschter Abweichung.
- Vorteil: keine **versteckten Kopplungen** über geteilte Referenzen, leichter zu testen.

8) Lesbare Daten-Pipelines: Komposition ohne Zwischenzustand

```
def normalisiere(mu, sigma):
    return (xs) -> mappe( x -> (x - mu) / sigma, xs )

def nur_positiv():
    return (xs) -> filtere( x -> x > 0, xs )

def total():
    return (xs) -> fold( (a,b) -> a + b, 0, xs )

def pipeline(xs, mu, sigma):
    return total()( nur_positiv()( normalisiere(mu, sigma)(xs) ) )
```

Erklärung:

- **Komposition** (∘) statt Zwischenspeicher: „erst normalisieren, dann filtern, dann summieren“.
- Jeder Schritt ist **pure** und **eigenständig testbar**.
- Die Pipeline beschreibt **was** passiert (Datenfluss), nicht **wie** iteriert wird.

9) Typische Prüfungsaufgaben im Flow

(a) Summe der positiven Quadrate

```
def summe_positive_quadrate(xs):
    return fold( (a,b) -> a + b, 0,
                 mappe( x -> x * x,
                        filtere( x -> x > 0, xs ) ) )
```

Erklärung:

- Erst **filter** (nur positive), dann **map** (quadrieren), dann **fold** (aufsummieren).
- Klar getrennte Verantwortung pro Schritt, keine Mutation.

(b) Nested Summe → siehe Abschnitt 4 (*summe_nested*). **(c) Zinseszins mit Beitrag** → siehe Abschnitt 5 (*zinseszins_mit_beitrag*). **(d) .txt sammeln & transformieren & sortiert ausgeben**

```
def txt_namen_sortiert(baum):
    return sortiert_immut(
             mappe( base_name_ohne_ext,
                    finde_txt(baum) ) )
```

Erklärung:

- **Kern** ist die pure Traversierung (*finde_txt*).
- Transformation (Basename) ist **pure**.
- **Sortieren ohne In-Place** liefert neue Liste → keine Nebenwirkung.

10) Prüfungs-Merker zum Formulieren

- „Ich nutze **rekursive Fallunterscheidung** (leer vs. Kopf+Rest) mit **Basisfall** für neutrale Elemente.“
- „**fold/map/filter** ersetzen Schleifen und Mutation; der **Akkumulator** wird als Wert weitergereicht.“
- „Die Funktionen sind **pure** (keine Seiteneffekte), Daten bleiben **immutable** (keine In-Place-Updates).“
- „**Komposition/Pipeline** beschreibt den Datenfluss klar und testbar.“

- „Bei IO-Problemen trenne ich **reine Logik** (Baum/Struktur) von den **Seiteneffekten** (Ein/Ausgabe)."