Informatik I: Einführung in die Programmierung

7.1 Entwurf von Schleifen, Hilfsfunktionen und Akkumulatoren

Albert-Ludwigs-Universität Freiburg

UNI FREIBURG

Peter Thiemann

20. November 2019

Entwurf von Schleifen
Fallstudie: Rechnen mit Polynomen
Skalarmultiplikation
Auswertung
Ableitung
Integration
Binäre Operationen
Addition
Multiplikation
Extra: Lexikographische Ordnung
Zusammenfassung

Entwurf von Schleifen

Entwurf von Schleifen
Fallstudie: Rechnen mit Polynomen
Skalarmultiplikation
Auswertung
Ableitung
Integration
Binäre Operationen
Addition
Multiplikation
Extra: Lexikographische Ordnung
Zusammenfassung

Fallstudie: Rechnen mit Polynomen

Polynome

UNI FREIBURG

Definition

Ein *Polynom vom Grad n* ist eine Folge von Zahlen $(a_0, a_1, \ldots, a_n)$, den *Koeffizienten*. Dabei ist $n \geq 0$ und $a_n \neq 0$.

Beispiele

- ()
- (1)
- $(3, 2, 1)$

Anwendungen

Kryptographie, fehlerkorrigierende Codes.

Entwurf von Schleifen
Fallstudie: Rechnen mit Polynomen
Skalarmultiplikation
Auswertung
Ableitung
Integration
Binäre Operationen
Addition
Multiplikation
Extra: Lexikographische Ordnung
Zusammenfassung

Rechenoperationen auf Polynomen

- (Skalar) Multiplikation mit einer Zahl c

$$c \cdot (a_0, a_1, \ldots, a_n) = (c \cdot a_0, c \cdot a_1, \ldots, c \cdot a_n)$$

- Auswertung an der Stelle x_0

$$(a_0, a_1, \ldots, a_n)[x_0] = \sum_{i=0}^{n} a_i \cdot x_0^i$$

- Ableitung

$$(a_0, a_1, \ldots, a_n)' = (1 \cdot a_1, 2 \cdot a_2, \ldots, n \cdot a_n)$$

- Integration

$$\int (a_0, a_1, \ldots, a_n) = (0, a_0, a_1/2, a_2/3, \ldots, a_n/(n+1))$$

Entwurf von Schleifen
Fallstudie: Rechnen mit Polynomen
Skalarmultiplikation
Auswertung
Ableitung
Integration
Binäre Operationen
Addition
Multiplikation
Extra: Lexikographische Ordnung
Zusammenfassung

Skalarmultiplikation

Skalarmultiplikation

UNI FREIBURG

Entwurf von Schleifen
Fallstudie: Rechnen mit Polynomen
Skalarmultiplikation
Auswertung
Ableitung
Integration
Binäre Operationen
Addition
Multiplikation
Extra: Lexikographische Ordnung
Zusammenfassung

$$c \cdot (a_0, a_1, \ldots, a_n) = (c \cdot a_0, c \cdot a_1, \ldots, c \cdot a_n)$$

Schritt 1: Bezeichner und Datentypen

Die Funktion `scalar_mult` nimmt als Eingabe

- `c : float`, den Faktor,
- `p : list`, ein Polynom.

Der Grad des Polynoms ergibt sich aus der Länge der Sequenz.

Skalarmultiplikation

UNI FREIBURG

Entwurf von Schleifen
Fallstudie: Rechnen mit Polynomen
Skalarmultiplikation
Auswertung
Ableitung
Integration
Binäre Operationen
Addition
Multiplikation
Extra: Lexikographische Ordnung
Zusammenfassung

Schritt 2: Funktionsgerüst

```
def scalar_mult(
        c : float,
        p : list # of float
        ) -> list: # of float
    # fill in, initialization
    for a in p:
        # fill in action for each element
    return
```

Skalarmultiplikation

UNI FREIBURG

Entwurf von Schleifen
Fallstudie: Rechnen mit Polynomen
Skalarmultiplikation
Auswertung
Ableitung
Integration
Binäre Operationen
Addition
Multiplikation
Extra: Lexikographische Ordnung
Zusammenfassung

Schritt 3: Beispiele

```
scalar_mult(42, []) == []
scalar_mult(42, [1,2,3]) == [42,84,126]
scalar_mult(-0.1, [1,2,3]) == [-0.1,-0.2,-0.3]
```

Skalarmultiplikation

UNI FREIBURG

Entwurf von Schleifen
Fallstudie: Rechnen mit Polynomen
Skalarmultiplikation
Auswertung
Ableitung
Integration
Binäre Operationen
Addition
Multiplikation
Extra: Lexikographische Ordnung
Zusammenfassung

Schritt 4: Funktionsdefinition

```
def scalar_mult(
        c : float,
        p : list # of float
        ) -> list: # of float
    result = []
    for a in p:
        result = result + [c * a]
    return result
```

Muster: Akkumulator

UNI FREIBURG

Rumpf der Skalarmultiplikation

```
result = []
for a in p:
    result = result + [c * a]
return result
```


Entwurf von Schleifen
Fallstudie: Rechnen mit Polynomen
Skalarmultiplikation
Auswertung
Ableitung
Integration
Binäre Operationen
Addition
Multiplikation
Extra: Lexikographische Ordnung
Zusammenfassung

Muster: Akkumulator

UNI FREIBURG

Rumpf der Skalarmultiplikation

```
result = []
for a in p:
    result = result + [c * a]
return result
```

Variable `result` ist Akkumulator

- In `result` wird das Ergebnis aufgesammelt
- `result` wird vor der Schleife initialisiert
- Jeder Schleifendurchlauf erweitert das Ergebnis in `result`

Entwurf von Schleifen
Fallstudie: Rechnen mit Polynomen
Skalarmultiplikation
Auswertung
Ableitung
Integration
Binäre Operationen
Addition
Multiplikation
Extra: Lexikographische Ordnung
Zusammenfassung

Auswertung

Auswertung

UNI FREIBURG

Entwurf von Schleifen
Fallstudie: Rechnen mit Polynomen
Skalarmultiplikation
Auswertung
Ableitung
Integration
Binäre Operationen
Addition
Multiplikation
Extra: Lexikographische Ordnung
Zusammenfassung

$$(a_0, a_1, \ldots, a_n)[x_0] = \sum_{i=0}^{n} a_i \cdot x_0^i$$

Schritt 1: Bezeichner und Datentypen

Die Funktion `poly_eval` nimmt als Eingabe

- `p : list`, ein Polynom,
- `x : float`, das Argument.

Der Grad des Polynoms ergibt sich aus der Länge der Sequenz.

Auswertung

UNI FREIBURG

Entwurf von Schleifen
Fallstudie: Rechnen mit Polynomen
Skalarmultiplikation
Auswertung
Ableitung
Integration
Binäre Operationen
Addition
Multiplikation
Extra: Lexikographische Ordnung
Zusammenfassung

Schritt 2: Funktionsgerüst

```
def poly_eval(
        p : list, # of float
        x : float
        ) -> float:
    # fill in
    for a in p:
        # fill in action for each element
    return
```

Auswertung

UNI FREIBURG

Entwurf von Schleifen
Fallstudie: Rechnen mit Polynomen
Skalarmultiplikation
Auswertung
Ableitung
Integration
Binäre Operationen
Addition
Multiplikation
Extra: Lexikographische Ordnung
Zusammenfassung

Schritt 3: Beispiele

```
poly_eval([], 2) == 0
poly_eval([1,2,3], 2) == 17
poly_eval([1,2,3], -0.1) == 0.83
```

Schritt 4: Funktionsdefinition

```
def poly_eval(
        p : list, # of float
        x : float
        ) -> float:
    result = 0
    i = 0
    for a in p:
        result = result + a * x ** i
        i = i + 1
    return result
```

Entwurf von Schleifen
Fallstudie: Rechnen mit Polynomen
Skalarmultiplikation
Auswertung
Ableitung
Integration
Binäre Operationen
Addition
Multiplikation
Extra: Lexikographische Ordnung
Zusammenfassung

Schritt 4: Alternative Funktionsdefinition

```
def poly_eval(
        p : list, # of float
        x : float
        ) -> float:
    result = 0
    for i, a in enumerate(p):
        result = result + a * x ** i
    return result
```

- `enumerate(seq)` liefert Paare (Laufindex, Element)
- Beispiel `list (enumerate([8, 8, 8])) == [(0, 8), (1, 8), (2, 8)]`

Entwurf von Schleifen
Fallstudie: Rechnen mit Polynomen
Skalarmultiplikation
Auswertung
Ableitung
Integration
Binäre Operationen
Addition
Multiplikation
Extra: Lexikographische Ordnung
Zusammenfassung

Ableitung

Ableitung

UNI FREIBURG

Entwurf von Schleifen
Fallstudie: Rechnen mit Polynomen
Skalarmultiplikation
Auswertung
Ableitung
Integration
Binäre Operationen
Addition
Multiplikation
Extra: Lexikographische Ordnung
Zusammenfassung

$$(a_0, a_1, \ldots, a_n)' = (1 \cdot a_1, 2 \cdot a_2, \ldots, n \cdot a_n)$$

Schritt 1: Bezeichner und Datentypen

Die Funktion `derivative` nimmt als Eingabe

- `p : list`, ein Polynom.

Der Grad des Polynoms ergibt sich aus der Länge der Sequenz.

Ableitung

UNI FREIBURG

Entwurf von Schleifen
Fallstudie: Rechnen mit Polynomen
Skalarmultiplikation
Auswertung
Ableitung
Integration
Binäre Operationen
Addition
Multiplikation
Extra: Lexikographische Ordnung
Zusammenfassung

Schritt 2: Funktionsgerüst

```
def poly_eval(
        p : list # of float
        ) -> list: # of float
    # fill in
    for a in p:
        # fill in action for each element
    return
```

Ableitung

UNI FREIBURG

Entwurf von Schleifen
Fallstudie: Rechnen mit Polynomen
Skalarmultiplikation
Auswertung
Ableitung
Integration
Binäre Operationen
Addition
Multiplikation
Extra: Lexikographische Ordnung
Zusammenfassung

Schritt 3: Beispiele

```
derivative([]) == []
derivative([42]) == []
derivative([1,2,3]) == [2,6]
```

Ableitung

UNI FREIBURG

Entwurf von Schleifen
Fallstudie: Rechnen mit Polynomen
Skalarmultiplikation
Auswertung
Ableitung
Integration
Binäre Operationen
Addition
Multiplikation
Extra: Lexikographische Ordnung
Zusammenfassung

Schritt 4: Funktionsdefinition

```
def derivative(
        p : list # of float
        ) -> list:
    result = []
    for i, a in enumerate(p):
        if i>0:
            result = result + [i * a]
    return result
```

Entwurf von Schleifen
Fallstudie: Rechnen mit Polynomen
Skalarmultiplikation
Auswertung
Ableitung
Integration
Binäre Operationen
Addition
Multiplikation
Extra: Lexikographische Ordnung
Zusammenfassung

Integration

Integration

UNI FREIBURG

$$\int (a_0, a_1, \ldots, a_n) = (0, a_0, a_1/2, a_2/3, \ldots, a_n/(n+1))$$

Schritt 1: Bezeichner und Datentypen

Die Funktion `integral` nimmt als Eingabe

- `p : list`, ein Polynom.

Der Grad des Polynoms ergibt sich aus der Länge der Sequenz.

Weitere Schritte

selbst

Entwurf von Schleifen
Fallstudie: Rechnen mit Polynomen
Skalarmultiplikation
Auswertung
Ableitung
Integration
Binäre Operationen
Addition
Multiplikation
Extra: Lexikographische Ordnung
Zusammenfassung

Entwurf von Schleifen
Fallstudie: Rechnen mit Polynomen
Skalarmultiplikation
Auswertung
Ableitung
Integration
Binäre Operationen
Addition
Multiplikation
Extra: Lexikographische Ordnung
Zusammenfassung

Binäre Operationen

Operationen mit zwei Polynomen

Entwurf von Schleifen
Fallstudie: Rechnen mit Polynomen
Skalarmultiplikation
Auswertung
Ableitung
Integration
Binäre Operationen
Addition
Multiplikation
Extra: Lexikographische Ordnung
Zusammenfassung

- Addition (falls $n \leq m$)

$$(a_0, a_1, \ldots, a_n) + (b_0, b_1, \ldots, b_m) = (a_0 + b_0, a_1 + b_1, \ldots, a_n + b_n, b_{n+1}, \ldots, b_m)$$

- Multiplikation von Polynomen

$$(a_0, a_1, \ldots, a_n) \cdot (b_0, b_1, \ldots, b_m) = (a_0 \cdot b_0, a_0 \cdot b_1 + a_1 \cdot b_0, \ldots, \sum_{i=0}^{k} a_i \cdot b_{k-i}, \ldots, a_n \cdot b_m)$$

Addition

Addition

UNI FREIBURG

$$(a_0, a_1, \ldots, a_n) + (b_0, b_1, \ldots, b_m)$$
$$= (a_0 + b_0, a_1 + b_1, \ldots, a_n + b_n, b_{n+1}, \ldots, b_m)$$

Schritt 1: Bezeichner und Datentypen

Die Funktion `poly_add` nimmt als Eingabe

- `p : list`, ein Polynom.
- `q : list`, ein Polynom.

Die Grade der Polynome ergeben sich aus der Länge der Sequenzen.

Achtung

Die Grade der Polynome können unterschiedlich sein!

Entwurf von Schleifen
Fallstudie: Rechnen mit Polynomen
Skalarmultiplikation
Auswertung
Ableitung
Integration
Binäre Operationen
Addition
Multiplikation
Extra: Lexikographische Ordnung
Zusammenfassung

Addition

UNI FREIBURG

Entwurf von Schleifen
Fallstudie: Rechnen mit Polynomen
Skalarmultiplikation
Auswertung
Ableitung
Integration
Binäre Operationen
Addition
Multiplikation
Extra: Lexikographische Ordnung
Zusammenfassung

Schritt 2: Funktionsgerüst

```
def poly_add(
        p : list, # of float
        q : list # of float
        ) -> list: # of float
    # fill in
    for i in range(...):
        # fill in action for each element
    return
```

Frage

Was ist . . . ?

Addition

UNI FREIBURG

Entwurf von Schleifen
Fallstudie: Rechnen mit Polynomen
Skalarmultiplikation
Auswertung
Ableitung
Integration
Binäre Operationen
Addition
Multiplikation
Extra: Lexikographische Ordnung
Zusammenfassung

Schritt 3: Beispiele

```
poly_add([], []) == []
poly_add([42], []) == [42]
poly_add([], [11]) == [11]
poly_add([1,2,3], [4,3,2,5]) == [5,5,5,5]
```

Addition

UNI FREIBURG

Entwurf von Schleifen
Fallstudie: Rechnen mit Polynomen
Skalarmultiplikation
Auswertung
Ableitung
Integration
Binäre Operationen
Addition
Multiplikation
Extra: Lexikographische Ordnung
Zusammenfassung

Schritt 3: Beispiele

```
poly_add([], []) == []
poly_add([42], []) == [42]
poly_add([], [11]) == [11]
poly_add([1,2,3], [4,3,2,5]) == [5,5,5,5]
```

Antwort

```
maxlen = max (len (p), len (q))
```

Addition

UNI FREIBURG

Entwurf von Schleifen
Fallstudie: Rechnen mit Polynomen
Skalarmultiplikation
Auswertung
Ableitung
Integration
Binäre Operationen
Addition
Multiplikation
Extra: Lexikographische Ordnung
Zusammenfassung

Schritt 4: Funktionsdefinition

```
def poly_add(
        p : list, # of float
        q : list # of float
        ) -> list: # of float
    maxlen = max (len (p), len (q))
    result = []
    for i in range(maxlen):
        result = result + [
            (p[i] if i < len(p) else 0) +
            (q[i] if i < len(q) else 0)]
    return result
```

Neuer Ausdruck

UNI FREIBURG

Bedingter Ausdruck (Conditional Expression)

```
exp_true if cond else exp_false
```

- Werte zuerst `cond` aus
- Falls Ergebnis kein Nullwert, dann werte `exp_true` als Ergebnis aus
- Sonst werte `exp_false` als Ergebnis aus

Beispiele

- `17 if True else 4 == 17`
- `"abc"[i] if i<3 else "␣"`

Entwurf von Schleifen
Fallstudie: Rechnen mit Polynomen
Skalarmultiplikation
Auswertung
Ableitung
Integration
Binäre Operationen
Addition
Multiplikation
Extra: Lexikographische Ordnung
Zusammenfassung

Addition

UNI FREIBURG

Schritt 4: Alternative Funktionsdefinition

```
def poly_add(
        p : list, # of float
        q : list # of float
        ) -> list: # of float
    maxlen = max (len (p), len (q))
    result = []
    for i in range(maxlen):
        ri = 0
        if i < len(p): ri = ri + p[i]
        if i < len(q): ri = ri + q[i]
        result = result + [ri]
    return result
```


Entwurf von Schleifen
Fallstudie: Rechnen mit Polynomen
Skalarmultiplikation
Auswertung
Ableitung
Integration
Binäre Operationen
Addition
Multiplikation
Extra: Lexikographische Ordnung
Zusammenfassung

Abstraktion — Hilfsfunktion

UNI FREIBURG

Beobachtung

- Code für Addition unübersichtlich, weil er mehrfach das gleiche Muster verwendet

```
1 if i < len(p): ri = ri + p[i]
2 p[i] if i < len(p) else 0
```

- Das gleiche Muster ist auch beim Produkt hilfreich...

⇒ Muster 2 in einer Hilfsfunktion abstrahieren!

Entwurf von Schleifen
Fallstudie: Rechnen mit Polynomen
Skalarmultiplikation
Auswertung
Ableitung
Integration
Binäre Operationen
Addition
Multiplikation
Extra: Lexikographische Ordnung
Zusammenfassung

Abstraktion — Hilfsfunktion

UNI FREIBURG

Entwurf von Schleifen
Fallstudie: Rechnen mit Polynomen
Skalarmultiplikation
Auswertung
Ableitung
Integration
Binäre Operationen
Addition
Multiplikation
Extra: Lexikographische Ordnung
Zusammenfassung

Beobachtung

- Code für Addition unübersichtlich, weil er mehrfach das gleiche Muster verwendet

```
1 if i < len(p): ri = ri + p[i]
2 p[i] if i < len(p) else 0
```

- Das gleiche Muster ist auch beim Produkt hilfreich...

⇒ Muster 2 in einer Hilfsfunktion abstrahieren!

Schritt 1: Bezeichner und Datentypen

Die Funktion `safe_index` nimmt als Eingabe

- `p : list` eine Sequenz
- `i : int` einen Index
- `d` einen Ersatzwert, der zu den Elementen von `p` passt

Sichere Indizierung

UNI FREIBURG

Entwurf von Schleifen
Fallstudie: Rechnen mit Polynomen
Skalarmultiplikation
Auswertung
Ableitung
Integration
Binäre Operationen
Addition
Multiplikation
Extra: Lexikographische Ordnung
Zusammenfassung

Schritt 2: Funktionsgerüst

```
def safe_index(
        p : list, # of T
        i : int,  # assume >= 0
        d         # of T, suitable for p
        ) -> list:
    # fill in
    return
```

Sichere Indizierung

UNI FREIBURG

Schritt 3: Beispiele

```
safe_index([1,2,3], 0, 0) == 1
safe_index([1,2,3], 2, 0) == 3
safe_index([1,2,3], 4, 0) == 0
safe_index([1,2,3], 4, 42) == 42
safe_index([], 0, 42) == 42
```


Entwurf von Schleifen
Fallstudie: Rechnen mit Polynomen
Skalarmultiplikation
Auswertung
Ableitung
Integration
Binäre Operationen
Addition
Multiplikation
Extra: Lexikographische Ordnung
Zusammenfassung

Sichere Indizierung

UNI FREIBURG

Schritt 3: Beispiele

```
safe_index([1,2,3], 0, 0) == 1
safe_index([1,2,3], 2, 0) == 3
safe_index([1,2,3], 4, 0) == 0
safe_index([1,2,3], 4, 42) == 42
safe_index([], 0, 42) == 42
```

Abstraktion des Musters

- Gefunden: `p[i] if i < len(p) else 0`
- Abstraktion: `p[i] if i < len(p) else d`

Entwurf von Schleifen
Fallstudie: Rechnen mit Polynomen
Skalarmultiplikation
Auswertung
Ableitung
Integration
Binäre Operationen
Addition
Multiplikation
Extra: Lexikographische Ordnung
Zusammenfassung

Schritt 4: Funktionsdefinition

```
def safe_index(
        p : list,  # of T
        i : int,   # assume >= 0
        d          # of T, suitable for p
        ) -> list:
    return p[i] if i < len(p) else d
```

oder gleichbedeutend

```
    if i < len(p):
        return p[i]
    else:
        return d
```

Entwurf von Schleifen
Fallstudie: Rechnen mit Polynomen
Skalarmultiplikation
Auswertung
Ableitung
Integration
Binäre Operationen
Addition
Multiplikation
Extra: Lexikographische Ordnung
Zusammenfassung

Addition

UNI FREIBURG

Funktionsdefinition mit Hilfsfunktion

```
def poly_add(
        p : list, # of float
        q : list # of float
        ) -> list: # of float
    maxlen = max (len (p), len (q))
    result = []
    for i in range(maxlen):
        result = result + [
            safe_index(p,i,0)
            + safe_index (q,i,0)]
    return result
```


Entwurf von Schleifen
Fallstudie: Rechnen mit Polynomen
Skalarmultiplikation
Auswertung
Ableitung
Integration
Binäre Operationen
Addition
Multiplikation
Extra: Lexikographische Ordnung
Zusammenfassung

Entwurf von Schleifen
Fallstudie: Rechnen mit Polynomen
Skalarmultiplikation
Auswertung
Ableitung
Integration
Binäre Operationen
Addition
Multiplikation
Extra: Lexikographische Ordnung
Zusammenfassung

Multiplikation

Multiplikation

UNI FREIBURG

$$(p_0, p_1, \ldots, p_n) \cdot (q_0, q_1, \ldots, q_m)$$
$$= (p_0 \cdot q_0, p_0 \cdot q_1 + p_1 \cdot q_0, \ldots, \sum_{i=0}^{k} p_i \cdot q_{k-i}, \ldots, p_n \cdot q_m)$$

Schritt 1: Bezeichner und Datentypen

Die Funktion `poly_mult` nimmt als Eingabe

- `p : list` ein Polynom
- `q : list` ein Polynom

und liefert als Ergebnis das Produkt der Eingaben.

Entwurf von Schleifen
Fallstudie: Rechnen mit Polynomen
Skalarmultiplikation
Auswertung
Ableitung
Integration
Binäre Operationen
Addition
Multiplikation
Extra: Lexikographische Ordnung
Zusammenfassung

Multiplikation

UNI FREIBURG

Schritt 2: Funktionsgerüst

```
def poly_mult(
        p : list, # of float
        q : list  # of float
        ) -> list: # of float
    # fill in
    for k in range(...):
        # fill in
        # compute k-th output element
    return
```


Entwurf von Schleifen
Fallstudie: Rechnen mit Polynomen
Skalarmultiplikation
Auswertung
Ableitung
Integration
Binäre Operationen
Addition
Multiplikation
Extra: Lexikographische Ordnung
Zusammenfassung

Multiplikation

UNI FREIBURG

Schritt 3: Beispiele

```
poly_mult([], []) == []
poly_mult([42], []) == []
poly_mult([], [11]) == []
poly_mult([1,2,3], [1]) == [1,2,3]
poly_mult([1,2,3], [0,1]) == [0,1,2,3]
poly_mult([1,2,3], [1,1]) == [1,3,5,3]
```


Entwurf von Schleifen
Fallstudie: Rechnen mit Polynomen
Skalarmultiplikation
Auswertung
Ableitung
Integration
Binäre Operationen
Addition
Multiplikation
Extra: Lexikographische Ordnung
Zusammenfassung

Multiplikation

UNI FREIBURG

Schritt 3: Beispiele

```
poly_mult([], []) == []
poly_mult([42], []) == []
poly_mult([], [11]) == []
poly_mult([1,2,3], [1]) == [1,2,3]
poly_mult([1,2,3], [0,1]) == [0,1,2,3]
poly_mult([1,2,3], [1,1]) == [1,3,5,3]
```

Beobachtungen

- Range `maxlen = len (p) + len (q) - 1`

Entwurf von Schleifen
Fallstudie: Rechnen mit Polynomen
Skalarmultiplikation
Auswertung
Ableitung
Integration
Binäre Operationen
Addition
Multiplikation
Extra: Lexikographische Ordnung
Zusammenfassung

Multiplikation

UNI FREIBURG

Schritt 4: Funktionsdefinition

```
def poly_mult(
        p : list, # of float
        q : list  # of float
        ) -> list: # of float
    result = []
    for k in range(len(p) + len(q) - 1):
        rk = ... # k-th output element
        result = result + [rk]
    return result
```


Entwurf von Schleifen
Fallstudie: Rechnen mit Polynomen
Skalarmultiplikation
Auswertung
Ableitung
Integration
Binäre Operationen
Addition
Multiplikation
Extra: Lexikographische Ordnung
Zusammenfassung

Das k-te Element

$$r_k = \sum_{i=0}^{k} p_i \cdot q_{k-i}$$

noch eine Schleife!

Entwurf von Schleifen
Fallstudie: Rechnen mit Polynomen
Skalarmultiplikation
Auswertung
Ableitung
Integration
Binäre Operationen
Addition
Multiplikation
Extra: Lexikographische Ordnung
Zusammenfassung

Multiplikation

UNI FREIBURG

Entwurf von Schleifen
Fallstudie: Rechnen mit Polynomen
Skalarmultiplikation
Auswertung
Ableitung
Integration
Binäre Operationen
Addition
Multiplikation
Extra: Lexikographische Ordnung
Zusammenfassung

Das k-te Element

$$r_k = \sum_{i=0}^{k} p_i \cdot q_{k-i}$$

noch eine Schleife!

Berechnung

```
rk = 0
for i in range(k+1):
    rk = rk + (safe_index(p,i,0)
               * safe_index(q,k-i,0))
```

Schritt 4: Funktionsdefinition, final

```
def poly_mult(
        p : list, # of float
        q : list  # of float
        ) -> list: # of float
    result = []
    for k in range(len(p) + len(q) - 1):
        rk = 0
        for i in range(k+1):
            rk = rk + (safe_index(p,i,0)
                       * safe_index(q,k-i,0))
        result = result + [rk]
    return result
```

Entwurf von Schleifen
Fallstudie: Rechnen mit Polynomen
Skalarmultiplikation
Auswertung
Ableitung
Integration
Binäre Operationen
Addition
Multiplikation
Extra: Lexikographische Ordnung
Zusammenfassung

Entwurf von Schleifen
Fallstudie: Rechnen mit Polynomen
Skalarmultiplikation
Auswertung
Ableitung
Integration
Binäre Operationen
Addition
Multiplikation
Extra: Lexikographische Ordnung
Zusammenfassung

Extra: Lexikographische Ordnung

Erinnerung: Lexikographische Ordnung

UNI FREIBURG

Entwurf von Schleifen
Fallstudie: Rechnen mit Polynomen
Skalarmultiplikation
Auswertung
Ableitung
Integration
Binäre Operationen
Addition
Multiplikation
Extra: Lexikographische Ordnung
Zusammenfassung

Gegeben

Zwei Sequenzen der Längen $m, n \geq 0$:

$$\vec{a} = "a_1 a_2 \ldots a_m" \qquad \vec{b} = "b_1 b_2 \ldots b_n"$$

Erinnerung: Lexikographische Ordnung

UNI FREIBURG

Entwurf von Schleifen
Fallstudie: Rechnen mit Polynomen
Skalarmultiplikation
Auswertung
Ableitung
Integration
Binäre Operationen
Addition
Multiplikation
Extra: Lexikographische Ordnung
Zusammenfassung

Gegeben

Zwei Sequenzen der Längen $m, n \geq 0$:

$$\vec{a} = "a_1 a_2 \ldots a_m" \qquad \vec{b} = "b_1 b_2 \ldots b_n"$$

$\vec{a} \leq \vec{b}$ in der lexikographischen Ordnung, falls

Es gibt $0 \leq k \leq \min(m, n)$, so dass

- $a_1 = b_1, \ldots, a_k = b_k$ und

  $$\vec{a} = "a_1 a_2 \ldots a_k a_{k+1} \ldots a_m" \qquad \vec{b} = "a_1 a_2 \ldots a_k b_{k+1} \ldots b_n"$$

  - $k = m$

    $$\vec{a} = "a_1 a_2 \ldots a_m" \qquad \vec{b} = "a_1 a_2 \ldots a_m b_{m+1} \ldots b_n"$$

  - oder $k < m$ und $a_{k+1} < b_{k+1}$.

Lexikographische Ordnung

UNI FREIBURG

Schritt 1: Bezeichner und Datentypen

Die Funktion `lex_ord` nimmt als Eingabe

- `a : list` eine Sequenz
- `b : list` eine Sequenz

und liefert als Ergebnis `True`, falls `a` $\leq$ `b`, sonst `False`.

Entwurf von Schleifen
Fallstudie: Rechnen mit Polynomen
Skalarmultiplikation
Auswertung
Ableitung
Integration
Binäre Operationen
Addition
Multiplikation
Extra: Lexikographische Ordnung
Zusammenfassung

Lexikographische Ordnung

UNI FREIBURG

Entwurf von Schleifen
Fallstudie: Rechnen mit Polynomen
Skalarmultiplikation
Auswertung
Ableitung
Integration
Binäre Operationen
Addition
Multiplikation
Extra: Lexikographische Ordnung
Zusammenfassung

Schritt 1: Bezeichner und Datentypen

Die Funktion `lex_ord` nimmt als Eingabe

- `a : list` eine Sequenz
- `b : list` eine Sequenz

und liefert als Ergebnis `True`, falls `a` $\leq$ `b`, sonst `False`.

Schritt 2: Funktionsgerüst

```
def lex_ord(
        a : list,
        b : list
        ) -> bool:
    # fill in
    for k in range(...):
        # fill in
    return
```

Lexikographische Ordnung

UNI FREIBURG

Schritt 3: Beispiele

```
lex_ord([], []) == True
lex_ord([42], []) == False
lex_ord([], [11]) == True
lex_ord([1,2,3], [1]) == False
lex_ord([1], [1,2,3]) == True
lex_ord([1,2,3], [0,1]) == False
lex_ord([1,2,3], [1,3]) == True
lex_ord([1,2,3], [1,2,3]) == True
```


Entwurf von Schleifen
Fallstudie: Rechnen mit Polynomen
Skalarmultiplikation
Auswertung
Ableitung
Integration
Binäre Operationen
Addition
Multiplikation
Extra: Lexikographische Ordnung
Zusammenfassung

Lexikographische Ordnung

UNI FREIBURG

Entwurf von Schleifen
Fallstudie: Rechnen mit Polynomen
Skalarmultiplikation
Auswertung
Ableitung
Integration
Binäre Operationen
Addition
Multiplikation
Extra: Lexikographische Ordnung
Zusammenfassung

Schritt 3: Beispiele

```
lex_ord([], []) == True
lex_ord([42], []) == False
lex_ord([], [11]) == True
lex_ord([1,2,3], [1]) == False
lex_ord([1], [1,2,3]) == True
lex_ord([1,2,3], [0,1]) == False
lex_ord([1,2,3], [1,3]) == True
lex_ord([1,2,3], [1,2,3]) == True
```

Beobachtungen

- Range `minlen = min (len (a), len (b))`

Lexikographische Ordnung

UNI FREIBURG

Schritt 4: Funktionsdefinition

```
def lex_ord(
        a : list,
        b : list
        ) -> bool:
    minlen = min (len (a), len (b))
    for k in range(minlen):
        if a[k] < b[k]:
            return True
        if a[k] > b[k]:
            return False
    # a is prefix of b or vice versa
    return len(a) <= len(b)
```


Entwurf von Schleifen
Fallstudie: Rechnen mit Polynomen
Skalarmultiplikation
Auswertung
Ableitung
Integration
Binäre Operationen
Addition
Multiplikation
Extra: Lexikographische Ordnung
Zusammenfassung

Zusammenfassung

Zusammenfassung

- Funktionen über Sequenzen verwenden for-in-Schleifen.

Zusammenfassung

- Funktionen über Sequenzen verwenden for-in-Schleifen.
- Ergebnisse werden meist in einer Akkumulator Variable berechnet.

Zusammenfassung

- Funktionen über Sequenzen verwenden for-in-Schleifen.
- Ergebnisse werden meist in einer Akkumulator Variable berechnet.
- Funktionen über mehreren Sequenzen verwenden for-range-Schleifen.

Zusammenfassung

- Funktionen über Sequenzen verwenden for-in-Schleifen.
- Ergebnisse werden meist in einer Akkumulator Variable berechnet.
- Funktionen über mehreren Sequenzen verwenden for-range-Schleifen.
- Der verwendete Range hängt von der Problemstellung ab.

Zusammenfassung

- Funktionen über Sequenzen verwenden for-in-Schleifen.
- Ergebnisse werden meist in einer Akkumulator Variable berechnet.
- Funktionen über mehreren Sequenzen verwenden for-range-Schleifen.
- Der verwendete Range hängt von der Problemstellung ab.
- Nicht-triviale Teilprobleme werden in Hilfsfunktionen ausgelagert.