Einführung in die Programmierung

Prof. Dr. Peter Thiemann
Marius Weidner
Simon Dorer, Timpe Hörig

Universität Freiburg
Institut für Informatik
Wintersemester 2025

Übungsblatt 13

Abgabe: Montag, 26.01.2026, 9:00 Uhr

Hinweis: Funktionen mit nur einem Ausdruck

In einigen Aufgaben auf diesem Blatt müssen Sie Funktionen definieren, deren Rumpf aus nur einem einzigen *Ausdruck* bestehen. Zum Beispiel:

```
def inc(x: int) -> int:
    return x + 1

def prepend[T](xs: list[T], x: T) -> list[T]:
    return [x] + xs
```

Für Funktionen höherer Ordnung verwendet man meist `lambda`-Funktionen innerhalb des Funktionsrumpfes, da diese in vielen Fällen besser zu lesen sind:

```
def add(x: int) -> Callable[[int], int]:
    return lambda y: x + y
```

Alternativ dürfen Sie aber auch normale Funktionen verwenden:

```
def add(x: int) -> Callable[[int], int]:
    def add_inner(y: int) -> int:
        return x + y
    return add_inner
```

Hinweis: Variablen und `lambda`-Funktionen

Wie in der Vorlesung erläutert, werden `lambda`-Funktionen typischerweise ohne Namen verwendet. Vermeiden Sie es daher Variablen eine anonyme Funktion zuzuweisen. Wenn Sie Funktionen mit einem Namen benötigen, greifen Sie stattdessen auf die übliche `def`-Syntax zurück.

```
# Nicht erlaubt:
inc: Callable[[int], int] = lambda x: x + 1

# Erlaubt:
def inc(x: int) -> int:
    return x + 1

# Auch erlaubt:
def add(x: int) -> Callable[[int], int]:
    return lambda y: x + y
```

Hinweis: n-stellige Funktionen

Eine n-stellige Funktion ist eine Funktion, die n Argumente entgegen nimmt. In den Beispielen oben ist `inc` eine einstellige und `prepend` zweistellige Funktion.

Reminder: Typannotationen

Denken Sie daran, den Typen immer so genau wie möglich anzugeben:

```python
# Unvollständig:
def get_age(persons: dict, name: str) -> int:
    return persons[name]


# Vollständig:
def get_age(persons: dict[str, int], name: str) -> int:
    return persons[name]


# Unvollständig:
def flip(t: tuple) -> tuple:
    return (t[1], t[0])

# Vollständig:
def flip[T, U](t: tuple[T, U]) -> tuple[U, T]:
    return (t[1], t[0])
```

Aufgabe 13.1 (Comprehensions; 6 Punkte; Datei: `comprehensions.py`)

In dieser Aufgabe sollen Sie Komprehensionen schreiben. Ihre Funktionsdefinitionen sollen dabei nur einen Ausdruck enthalten und **keine zusätzlichen (lokalen) Funktionsdefinitionen enthalten**. Verwenden Sie also insbesondere nicht die `filter`, `map` oder `reduce` Funktion.

(a) `cartesian`; **2 Punkte**

Schreiben Sie eine Funktion `cartesian`, die zwei Listen `xs` und `ys` als Argumente nimmt und einen Iterator zurückgibt, der alle Tupel des kartesischen Produkts von `xs` und `ys` liefert. Das kartesische Produkt enthält alle möglichen Tupel `(x, y)`, wobei `x` $\in$ `xs` und `y` $\in$ `ys` ist.

```
>>> list(cartesian([], []))
[]
>>> list(cartesian([1, 2], []))
[]
>>> list(cartesian([True, False], ["Hallo", "Welt", ":)"]))
[(True, 'Hallo'), (True, 'Welt'), (True, ':)'), (False, 'Hallo'),
↪  (False, 'Welt'), (False, ':)')]
```

(b) `divisible`; **2 Punkte**

Schreiben Sie eine Funktion `divisible`, die eine Liste ganzer Zahlen `xs` als Argument erhält. Die Funktion soll ein Dictionary zurückgeben, das für jedes

Element `x` ∈ `xs` auf die Menge aller Elemente `xi` ∈ `xs` abbildet, für die gilt, dass `x` durch `xi` ohne Rest teilbar ist.

```
>>> divisible([])
{}
>>> divisible([1, 2, 3])
{1: {1}, 2: {1, 2}, 3: {1, 3}}
>>> divisible([2, 4, 6, 8])
{2: {2}, 4: {2, 4}, 6: {2, 6}, 8: {8, 2, 4}}
>>> divisible([3, 5, 15, 30])
{3: {3}, 5: {5}, 15: {3, 5, 15}, 30: {3, 5, 30, 15}}
```

(c) `chunk`; **2 Punkte**
Schreiben Sie eine Funktion `chunk`, die eine Liste `xs` und eine natürliche Zahl `n` als Argumente nimmt und eine Liste von Teillisten zurückgibt. Jede Teilliste soll dabei aus `n` aufeinanderfolgenden Elementen aus `xs` bestehen. Berücksichtigen Sie dabei, dass die letzte Teilliste nur dann übernommen werden soll, falls sie genau `n` Elemente enthält, andernfalls soll sie verworfen werden.

```
>>> chunk([], 3)
[]
>>> chunk([1, 2, 3, 4], 2)
[[1, 2], [3, 4]]
>>> chunk([1, 2, 3, 4, 5, 6, 7], 2)
[[1, 2], [3, 4], [5, 6]]
>>> chunk(["a", "b", "c", "d", "e"], 3)
[['a', 'b', 'c']]
```

Aufgabe 13.2 (Funktionale Programmierung; 6 Punkte; Datei: `functional.py`)

In dieser Aufgabe sollen alle Ihre Funktionsdefinitionen nur einen Ausdruck enthalten. **Zudem dürfen Sie *keine* Komprehensionen benutzen.**

(a) `sum_even_squares`; **2 Punkte**
Schreiben Sie eine Funktion `sum_even_squares`, die eine Liste `xs` von ganzen Zahlen als Argument nimmt und die Summe der Quadratzahlen von jedem geraden Element zurückgibt.

```
>>> sum_even_squares([])
0
>>> sum_even_squares([2, 3])
4
>>> sum_even_squares([1, 2, 3, 4, 5, 6])
56
```

(b) `is_additive`; **2 Punkte**
Schreiben Sie eine Funktion `is_additive`, die zwei Listen von ganzen Zahlen `xs` $= [x_0, \ldots, x_n]$ und `ys` $= [y_0, \ldots, y_n]$ als Argumente nimmt. Zurückgegeben werden soll eine Funktion, die eine beliebige einstellige Funktion auf ganzen Zahlen als Argument nimmt und zurückgibt, ob diese Funktion die additive Cauchy-

Funktionalgleichung für alle Paare (x_i, y_i) mit $i = 0, \ldots, n$ erfüllt. Eine Funktion erfüllt genau dann die additive Cauchy-Funktionalgleichung, wenn gilt: `f(x + y) == f(x) + f(y)`. Sie dürfen davon ausgehen, dass `len(xs) == len(ys)`.

```
>>> check = is_additive([1, 2, 3], [2, 3, 5])
>>> check(lambda x: x**2)
False
>>> check(lambda x: x * 2)
True
```

(c) `inverse`; **2 Punkte**
Schreiben Sie eine Funktion `inverse`, die eine einstellige Funktion `f` und eine Liste `domain_space` als Argumente nimmt. Sie dürfen dabei annehmen, dass `f` bijektiv mit Domain `domain_space` ist. Zurückgeben soll `inverse` das inverse von `f`, also eine Funktion die für jedes `y` ∈ Codomain, ein `x` ∈ Domain zurückgibt, sodass `f(x) == y`.

```
>>> to_int = lambda x: 1 if x is True else 0
>>> domain = [False, True]
>>> codomain = [0, 1]
>>> [to_int(x) for x in domain]
[0, 1]
>>> to_bool = inverse(to_int, domain)
>>> [to_bool(y) for y in codomain]
[False, True]
```

Aufgabe 13.3 (Fold; 8 Punkte; Datei: `fold.py`)

Sie haben Funktionen wie `sum` oder `all` kennengelernt. Diese Funktionen haben eins gemeinsam: Sie nehmen ein iterierbares Objekt, wie zum Beispiel eine Liste und *falten* dieses zu einem einzelnen Wert zusammen. Wie die Werte zusammengefaltet werden, unterscheidet sich bei `sum` und `all` natürlich. Bei beiden kann man aber gleich vorgehen: Man wählt einen Startwert bzw. einen Zählwert und eine zweistellige Funktion. Solange es noch ein Element in der Liste gibt, wird die Funktion auf diesem Element und dem Zählwert angewandt und das Ergebnis im Zählwert gemerkt. Gibt es keine Elemente mehr, wird der Zählwert zurückgegeben.

Abhängig vom Startwert und der Funktion, kann es einen Unterschied[1] machen, ob man die Liste von links nach rechts (`foldl`[2]) oder von rechts nach links (`foldr`) zusammenfaltet.

Betrachten wir nun ein Beispiel für die Faltung über Listen mit folgenden Werten:

```
>>> f = lambda x, y: x - y
>>> start = 0
>>> xs = [1, 2, 3]
```

[1] Falls `f` eine assoziative und kommutative Operation ist und `start` ein neutrales Element bzgl. dieser Operation ist, berechnen diese Funktionen den gleichen Wert.

[2] wie `reduce` aus dem Modul `functools`

Die Faltungen von links nach rechts und von rechts nach links lassen sich dann wie folgt berechnen:

```
>>> foldr(f, start, xs) == f(1, f(2, f(3, start))) == (1 - (2 - (3 -
↪  0))) == 2
True
>>> foldl(f, start, xs) == f(f(f(start, 1), 2), 3) == (((0 - 1) - 2) -
↪  3) == -6
True
```

(a) `foldr`; **2 Punkte**
Implementieren Sie die Funktion `foldr`, die eine zweistellige Funktion `f`, einen Startwert `start` und eine Liste `xs` als Argumente nimmt, die Liste `xs` von *rechts nach links* mithilfe von `f` zusammenfaltet und das Ergebnis zurückgibt. Ein halber Punkt wird für die richtige Typannotation mit Typvariablen vergeben. Sie dürfen keine `for`- oder `while`-Schleifen sowie `if`-Statements benutzen. **Lösen Sie die Aufgabe stattdessen mithilfe von Pattern-Matching und Rekursion.**

```
>>> foldr(lambda x, _: x, 0, xs)
1
>>> foldr(lambda _, y: y, 0, xs)
0
>>> foldr(lambda x, y: x + y / 2, 0, xs)
2.75
>>> foldr(lambda x, y: [x] + y, [5], xs)
[1, 2, 3, 5]
```

(b) `foldl`; **2 Punkte**
Implementieren Sie die Funktion `foldl`, die eine zweistellige Funktion `f`, einen Startwert `start` und eine Liste `xs` als Argumente nimmt, die Liste `xs` von *links nach rechts* mithilfe von `f` zusammenfaltet und das Ergebnis zurückgibt. Ein halber Punkt wird für die richtige Typannotation mit Typvariablen vergeben. Sie dürfen keine `for`- oder `while`-Schleifen sowie `if`-Statements benutzen. **Lösen Sie die Aufgabe stattdessen mithilfe von Pattern-Matching und Rekursion.**

```
>>> foldl(lambda x, _: x, 0, xs)
0
>>> foldl(lambda _, y: y, 0, xs)
3
>>> foldl(lambda x, y: x + y / 2, 0, xs)
3.0
>>> foldl(lambda x, y: x + [y], [5], xs)
[5, 1, 2, 3]
```

(c) `all`; **2 Punkte**
Implementieren Sie die Funktion `all`, die eine Liste `xs` von Wahrheitswerten als

Argument nimmt und mithilfe von `foldl` oder `foldr` berechnet und zurückgibt, ob alle Wahrheitswerte der Liste wahr (`True`) sind. Ihre Funktionsdefinition soll dabei aus nur einem Ausdruck bestehen (siehe Hinweis oben).

```
>>> all([])
True
>>> all([True, True, False, True])
False
>>> all(10 * [True])
True
```

(d) `pot`; **2 Punkte**
Implementieren Sie die Funktion `pot`, die eine Liste `xs` $= [x_0, x_1, x_2, \ldots, x_n]$ von ganzen Zahlen als Argument nimmt und mithilfe von `foldl` oder `foldr` die rechts assoziativ Potenz $x_0 ** (x_1 ** (x_2 ** \ldots ** x_n))$ dieser Zahlen berechnet und zurückgibt. Ihre Funktionsdefinition soll dabei aus nur einem Ausdruck bestehen (siehe Hinweis oben).

Hinweis: Komplexe Zahlen werden in Python als `n+mj` dargestellt, wobei `n` $\in \mathbb{R}$ der reale, `m` $\in \mathbb{R}$ der imaginäre Teil und `j` die imaginäre Einheit ist. Ein solcher Ausdruck hat den Typ `complex`, wobei `float` ein Subtyp von `complex` ist.

```
>>> pot([])
1
>>> pot([2, 3, 4])
241785163922925834941235 2
>>> pot([1, 10, 2, 3, 1, 20])
1
>>> pot([-2, 2, -1])
(8.659560562354934e-17+1.4142135623730951j)
```

(e) `map`, `filter`; **0 Punkte** (Knobelaufgabe)
Die Funktionen `map` und `filter` können wir nun auch analog mit unseren selbst definierten Faltungs-Funktionen implementieren. Diese nehmen jeweils eine einstellige Funktion `f` und eine Liste `xs` als Argumente. `map` gibt eine neue Liste mit den von `f` abgebildeten Werten zurück. `filter` gibt eine neue Liste mit den Werten von `xs` zurück, für die `f` wahr ist. Schaffen Sie es, die Funktionen jeweils mit einem einzigen Aufruf von `foldl` oder `foldr` und ohne Rekursion zu schreiben?

```
>>> map(lambda x: -x, xs)
[-1, -2, -3]
>>> map(lambda x: -x, [])
[]
>>> filter(lambda x: x > 2, xs)
[3]
>>> filter(lambda x: x < 0, xs)
```

```
[]
>>> filter(lambda x: x < 0, [])
[]
```

Aufgabe 13.4 (Erfahrungen; 0 Punkte; Datei: `NOTES.md`)

Notieren Sie Ihre Erfahrungen mit diesem Übungsblatt (benötigter Zeitaufwand, Probleme, Bezug zur Vorlesung, Interessantes, etc.).

Editieren Sie hierzu die Datei `NOTES.md` im Abgabeordner dieses Übungsblattes auf unserer Webplatform. Halten Sie sich an das dort vorgegebene Format, da wir den Zeitbedarf mit einem Python-Skript automatisch statistisch auswerten. Die Zeitangabe `7.5 h` steht dabei für 7 Stunden 30 Minuten.