Einführung in die Programmierung

Prof. Dr. Peter Thiemann
Marius Weidner, Hannes Saffrich
Simon Dorer, Sebastian Klähn

Universität Freiburg
Institut für Informatik
Wintersemester 2024

Übungsblatt 12

Abgabe: Montag, 20.01.2024, 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.

Aufgabe 12.1 (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) `only_positive`; **2 Punkte**

Schreiben Sie eine Funktion `only_positive`, die eine Liste `xs` von Ganzzahlen als Argument nimmt und genau dann True zurückgibt, wenn `xs` keine negativen Zahlen enthält.

```
>>> only_positive([1, 2, 3, 4, 5, -1])
False
>>> only_positive([1, 2, 3, 4])
True
>>> only_positive([])
True
```

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

Schreiben Sie eine Funktion `sum_even_squares`, die eine Liste `xs` von Ganzzahlen als Argument nimmt und die Summe der Quadratzahlen von jedem geraden Element zurück gibt.

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

(c) `extract_prefixes`; **2 Punkte**

In real-world Projekten kommt es häufiger vor, dass man komplexe Datentypen auf einfache Werte reduzieren will. In dieser Aufgabe bekommen Sie eine Liste von Tupeln (bool, str). Die Funktion `extract_prefixes` soll zuerst überprüfen, ob das erste (bool) Argument `True` ist. Falls ja, soll das Präfix der E-Mail-Adresse (alles bis zum ersten '@') des zweiten Felds extrahiert werden. Schlussendlich soll eine String zurückgegeben werden, der alle extrahierten Prefixe als komma-seperierte Liste enthält.
Tipp: verwenden Sie `split, filter` und `join`.

```
>>> extract_prefixes([
...     (True, "max@mustermann.de"),
...     (False, "franka2001@musterfrau.org"),
...     (True, "jürgen@uni-erlangen.edu")
... ])
'max,jürgen'
```

```
>>> extract_prefixes([])
''
```

Aufgabe 12.2 (Comprehensions; 6 Punkte; Datei: `comprehensions.py`)

In dieser Aufgabe sollen Sie Komprehensionen schreiben. Ihre Funktionsdefinitionen sollen dabei nur einen Ausdruck enthalten. **Verwenden Sie nicht die `filter`, `map` oder `reduce`-Funktionen der Standard-Library.**

(a) `flatten2D`; **2 Punkte**

Schreiben Sie eine Funktion `flatten2D`, die eine Liste von Listen `ls` und eine natürliche Zahl `n` als Argumente nimmt und eine neue Liste zurückgibt, die alle Elemente der inneren Listen von `ls` enthält, die mindestens `n` Elemente haben:

```
>>> flatten2D([], 0)
[]
>>> flatten2D([[], []], 1337)
[]
>>> flatten2D([[1, 2], [3, 4, 5], [], [6, 7, 8, 9]], 3)
[3, 4, 5, 6, 7, 8, 9]
>>> flatten2D([["Hallo", "Welt", "!"] , ["HelloWorld"],
↪  ["Bonjour", "le monde"]], 2)
['Hallo', 'Welt', '!', 'Bonjour', 'le monde']
```

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

Schreiben Sie eine Funktion `dot_product`, die zwei Vektoren `a` und `b` (Listen von Gleitkommazahlen) als Argumente nimmt und das Skalarprodukt der beiden Vektoren berechnet. Das Skalarprodukt zweier Vektoren $a, b \in \mathbb{R}^n$ ist wie folgt definiert:

$$\begin{bmatrix} a_1 \\ a_2 \\ \vdots \\ a_n \end{bmatrix} \times \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{bmatrix} = \sum_{i=1}^{n} a_i \cdot b_i = a_1 \cdot b_1 + a_2 \cdot b_2 + \ldots + a_n \cdot b_n$$

Sie dürfen davon ausgehen, dass die beiden Vektoren `a` und `b` gleich lang sind. Vermeiden Sie in Ihrer Implementierung das Erzeugen einer temporären Liste.

```
>>> dot_product([], [])
0
>>> dot_product([-3.3], [4.7])
-15.51
>>> dot_product([1., -2., 3., 4.], [-5., 6., 7., 8.])
36.0
```

(c) `filter_map`; **2 Punkte**

Schreiben Sie eine Funktion `filter_map`, die zwei Funktionen `f` und `g` sowie eine Liste `xs` als Argumente erhält. Die Funktion soll eine Liste zurückgeben, in der `f` auf alle Elemente von `xs` angewendet wird, für die `g`, angewandt auf

das jeweilige Element, `True` liefert. Elemente, für die `g` `False` zurückgibt, sollen aus der Liste herausgefiltert werden.

```
>>> filter_map(lambda x: -x, lambda x: x < 3, [1, 3, 2, 4])
[-1, -2]
>>> filter_map(lambda x: -x, lambda x: x < 1, [1, 3, 2, 4])
[]
```

Aufgabe 12.3 (Fold; 8 Punkte; Datei: `fold.py`)

Sie haben Funktionen wie `sum` oder `any` 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 `any` 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 den Zählwert angewandt und das Ergebnis im Zählwert gemerkt. Gibt es keine Elemente mehr, wird der Zählwert zurückgeben.

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.

Um diese Funktionen zu implementieren, verwenden wir die in der Funktionalen Programmierung übliche Unärbaum-Darstellung für Listen `LList[T]`, die wie folgt definiert ist:

```
@dataclass
class Cons[T]:
    head: T
    tail: 'LList[T]' = None

type LList[T] = Optional[Cons[T]]
```

Leere Listen werden in dieser Darstellung mit `None` dargestellt. Instanzen von `Cons[T]` repräsentieren (Sub-) Listen, die aus mindestens einem Element (`head`) bestehen und zusätzlich einen optionalen Verweis auf auf die nächste Subliste (`tail`) haben. Die Python-Liste `[]` wird wird in dieser Darstellung als `None`, `[-42]` als `Cons(-42)` und `[1, 2, 3]` als `Cons(1, Cons(2, Cons(3)))` dargestellt.

Betrachten wir nun ein Beispiel für Faltung mit folgenden Werten:

```
>>> f = lambda x, y: x - y
>>> start = 0
>>> xs = Cons(1, Cons(2, Cons(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 LList `xs` als Argumente nimmt, die Liste `xs` von *rechts nach links* mithilfe von `f` zusammenfaltet und das Ergebnis zurückgibt. 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 LList `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) `sum`; **2 Punkte**
Implementieren Sie die Funktion `sum`, die eine LList `xs` von ganzen Zahlen als Argument nimmt und mithilfe von `foldr` oder `foldl` die Summe dieser Zahlen

berechnet und zurückgibt. Schreiben Sie die Funktion als Variable mit Funktionswert oder als Funktion, die aus einer einzigen `return`-Anweisung besteht, wie im Hinweis oben gezeigt.

```
>>> sum(xs)
6
>>> sum(None)
0
```

(d) `any`; **2 Punkte**
Implementieren Sie die Funktion `any`, die eine LList `xs` von Wahrheitswerten als Argument nimmt und mithilfe von `foldr` oder `foldl` berechnet und zurückgibt, ob mindestens ein Wahrheitswert `True` ist. Schreiben Sie die Funktion als Variable mit Funktionswert oder als Funktion, die aus einer einzigen `return`-Anweisung besteht, wie im Hinweis oben gezeigt.

```
>>> any(Cons(False, Cons(False, Cons(True, Cons(False)))))
True
>>> any(None)
False
```

(e) `map`, `filter`; **0 Punkte** (Knobelaufgabe)
Die Funktion `map` wendet eine einstellige Funktion auf jedes Element des angegebenen iterierbaren Objekts an und produziert der Reihe nach diese Funktionswerte. Die Funktion `filter` wendet eine einstellige Funktion auf jedes Element des gegebenen iterierbaren Objekts an und produziert die Elemente, bei denen der Funktionswert wahr ist. Beispielsweise:

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

Die Funktionen `map` und `filter` können wir nun auch analog für `LList` definieren. Diese nehmen jeweils eine einstellige Funktion `f` und eine `LList xs` als Argumente. `map` gibt eine neue `LList` mit den von `f` abgebildeten Werten zurück. `filter` gibt eine neue `LList` 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)
Cons(head=-1, tail=Cons(head=-2, tail=Cons(head=-3, tail=None)))
>>> map(lambda x: -x, None)  # None
>>> filter(lambda x: x > 2, xs)
Cons(head=3, tail=None)
>>> filter(lambda x: x < 0, xs)   # None
>>> filter(lambda x: x < 0, None)   # None
```

Aufgabe 12.4 (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.