Einführung in die Programmierung

Prof. Dr. Peter Thiemann
Marius Weidner, Hannes Saffrich
Lukas Kleinert, Timpe Hörig

Universität Freiburg
Institut für Informatik
Wintersemester 2023

Übungsblatt 12

Abgabe: Montag, 22.01.2024, 9:00 Uhr morgens

Hinweis

In einigen Aufgaben auf diesem Blatt müssen Sie Funktionen definieren, deren Rumpf aus einem einzigen Ausdruck besteht - wie üblich in der Funktionalen Programmierung. Zum Beispiel:

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

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

Funktionen, die keine Typvariablen verwenden, können Sie auch als Variable mit Funktionswert definieren:

```
from typing import Callable

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

Diese Schreibweise ist insbesondere bei bestimmten verschachtelten Funktionen angenehmer zu lesen:

```
def mul_1(x: int) -> Callable[[int], int]:
    def mul_with_x(y):
        return x * y
    return mul_with_x

mul_2: Callable[[int], Callable[[int], int]] = lambda x: lambda y: x * y

assert mul_1(2)(3) == 6
assert mul_2(2)(3) == 6
```

Hinweis

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

Aufgabe 12.1 (Funktionale Programmierung; 6 + 0.5 Punkte; Datei: `functional.py`)
In dieser Aufgabe sollen alle Ihre Funktionsdefinition bis auf eine `return`-Anweisung keine weiteren Zeilen enthalten. Zudem dürfen Sie *keine* Komprehensionen benutzen.

(a) `has_negative`; **2 Punkte**
Schreiben Sie eine Funktion `has_negative`, die eine Liste `xs` von Ganzzahlen als Argument nimmt und genau dann True zurückgibt, wenn `xs` mindestens eine negative Zahl enthält.

```
assert has_negative([1, 2, 3, 4, 5, -1])
assert not has_negative([1, 2, 3, 4])
assert not has_negative([])
```

(b) `factorial`; **2 Punkte**
Schreiben Sie eine *nicht-rekursive* Funktion `factorial`, die eine Ganzzahl `n` als Argument nimmt und die Fakultät von `n` zurückgibt. Ist `n` kleiner als 1, soll 1 zurückgeben werden.

```
assert [factorial(n) for n in range(7)] == [1, 1, 2, 6, 24, 120, 720]
assert factorial(-1) == 1
```

(c) `compose`; **2 + 0.5 Punkte**
Schreiben Sie eine Funktion `compose`, die zwei einstellige Funktionen `f` und `g` als Argumente nimmt und die Funktionskomposition `f` ∘ `g` zurückgibt.

```
add2 = lambda x: x + 2
mul5 = lambda x: x * 5
add2mul5 = compose(mul5, add2)
assert add2mul5(3) == 25
longest = compose(max, functools.partial(map, len))
assert longest(["abc", "de", "fghijkl"]) == 7
```

Ein halber Punkt wird für die richtige Typannotation mit Typvariablen vergeben. Beachten Sie, dass `f` und `g` beliebige Funktionen sein können, die lediglich durch die Komposition eine Einschränkung im Typ erhalten.

Aufgabe 12.2 (Komprehensionen; 4 + 0.5 Punkte; Datei: `comprehensions.py`)
In dieser Aufgabe sollen Sie (Listen-) Komprehensionen schreiben. Die Funktionen sollen auch hier bis auf eine `return`-Anweisung keine weiteren Zeilen enthalten. Verwenden Sie auch *nicht* `filter`, `map` oder `reduce`.

(a) `quad_even`; **2 Punkte**
Schreiben Sie eine Funktion `quad_even`, die eine ganze Zahl `n` als Argument nimmt und eine Liste von allen Quadratzahlen zurückgibt, die kleiner oder gleich `n` und gerade sind. Sie dürfen davon ausgehen, dass `n` nicht negativ ist.

```
assert quad_even(99) == [4, 16, 36, 64]
assert quad_even(100) == [4, 16, 36, 64, 100]
assert quad_even(0) == []
```

(b) `filter_map`; **2 + 0.5 Punkte**
Schreiben Sie eine Funktion `filter_map`, die zwei Funktion `f` und `g` und eine Liste `xs` als Argumente nimmt. Es soll eine Liste zurückgegeben werden, bei der die Funktion `f` auf alle Elemente von `xs` angewandt wurde[1], für die die Funktion `g` angewandt auf das Element `True` ist. Elemente für die `g False` zurückgibt, sollen rausgefiltert werden. Ein halber Punkt wird für die richtige Typannotation mit Typvariablen vergeben.

```
lt3 = lambda x: x < 3
neg = lambda x: -x
assert filter_map(neg, lt3, [1, 3, 2, 4]) == [-1, -2]
lt1 = lambda x: x < 1
assert filter_map(neg, lt1, [1, 3, 2, 4]) == []
```

(c) `group_by`; **0 Punkte** (Knobelaufgabe)
Schreiben Sie eine Funktion `group_by`, die eine Funktion `f` und eine Menge `s` als Argumente nimmt. Die Funktion soll ein Dictionary zurückgeben, das jedes Element `el` von `s` dem Funktionswert `f(el)` (als Menge) zuweist.

```
s = {1, 15, 4, 8, 33, 5, 23, 4, 5}
mod4 = lambda x: x % 4
assert group_by(mod4, s) == {
    1: {1, 5, 33}, 3: {23, 15}, 0: {8, 4}
}
assert group_by(lambda x: x, set()) == {}
```

Aufgabe 12.3 (Fold; 8 + 1 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[2] machen, ob man die Liste von Links nach Rechts (`foldl`, so wie `reduce` aus dem Modul `functools`) oder von Rechts nach Links (`foldr`) zusammenfaltet.
Hier ein Beispiel für `xs = [1,2,3,4,5]`, `start = 0` und `f = (lambda x,y: x-y)`:

```
foldr(f, start, xs) = f(1, f(2, f(3, f(4, f(5, start)))))
    = 1 - (2 - (3 - (4 - (5 - 0)))) = 3
foldl(f, start, xs) = f(f(f(f(f(start, 1), 2), 3), 4), 5)
    = ((((0 - 1) - 2) - 3) - 4) - 5 = -15
```

[1] Der Eingabetyp und Rückgabetyp von `f` kann unterschiedlich sein.
[2] Falls `f` eine assoziative und kommutative Operation ist und `start` ein neutrales Element bzgl. dieser Operation ist, berechnen diese Funktionen den gleichen Wert.

In Übungsblatt 7 haben Sie die Datenstruktur `LList` implementiert, die eine andere Art ist, Listen darzustellen. Diese Darstellung einer Liste (bzw. eines Unärbaums) ist besonders kompatibel mit funktionaler Programmierung.

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

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

(a) `foldr`; **2 + 0.5 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. 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.

```
example = Cons(1, Cons(2, Cons(3, Cons(4))))
assert foldr(lambda x, _: x, 0, example) == 1
assert foldr(lambda _, y: y, 0, example) == 0
assert foldr(lambda x, y: x + y / 2, 0, example) == 3.25
assert foldr(lambda x, y: [x] + y, [5], example) == [1, 2, 3, 4, 5]
```

(b) `foldl`; **2 + 0.5 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.

```
assert foldl(lambda x, _: x, 0, example) == 0
assert foldl(lambda _, y: y, 0, example) == 4
assert foldl(lambda x, y: x + y / 2, 0, example) == 5.0
assert foldl(lambda x, y: x + [y], [5], example) == [5, 1, 2, 3, 4]
```

(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.

```
assert sum(example) == 10
assert 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.

```
assert any(Cons(False, Cons(False, Cons(True, Cons(False)))))
assert not any(None)
```

(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

```
assert list(map(lambda x: -x, [1, 2, 3, 4])) == [-1, -2, -3, -4]
assert list(filter(lambda x: x > 2, [1, 2, 3, 4])) == [3, 4]
```

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?

```
assert (map(lambda x: -x, example)
        == Cons(-1, Cons(-2, Cons(-3, Cons(-4)))))
assert map(lambda x: -x, None) is None  # type: ignore
assert filter(lambda x: x > 2, example) == Cons(3, Cons(4))
assert filter(lambda x: x < 0, example) is None
assert filter(lambda x: x < 0, None) is None  # type: ignore
```

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.