Informatik I: Einführung in die Programmierung

Prof. Dr. Peter Thiemann
Hannes Saffrich, Simon Ging
Wintersemester 2021

Universität Freiburg
Institut für Informatik

Übungsblatt 13

Abgabe: Montag, 31.01.2021, 9:00 Uhr morgens

Hinweis

Es gelten die selben Regeln wie bisher. Diese können in Blatt 8 eingesehen werden.

Hinweis

In diesem Übungsblatt müssen Sie **keine** Typannotationen schreiben.

Hinweis

In den meisten Aufgaben müssen Sie Funktionen definieren, deren Rumpf aus einem einzigen Ausdruck besteht - wie üblich in der Funktionalen Programmierung. Beispiel:

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

Diese Funktionen 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
```

Der Linter beschwert sich bei dieser Schreibweise mit

> "Do not assign a lambda expression, use a def."

Diese Meldung dürfen Sie ignorieren.

Aufgabe 13.1 (Funktionskomposition; Datei: `compose.py`; 2 Punkte)

In dieser Aufgabe sollen Ihre Funktionsdefinitionen, außer einer `return`-Anweisung, keine weiteren Anweisungen enthalten, oder wie im Hinweis als Variable mit Funktionswert definiert werden.

Schreiben Sie eine Funktion `compose`, die zwei einstellige Funktionen `f` und `g` als Argument nimmt und die Funktionskomposition $\texttt{f} \circ \texttt{g}$ zurückgibt.

Beispiel:

```
inc  = lambda x: x + 1
mul2 = lambda x: x * 2
assert compose(inc, mul2)(4) == 9
```

Aufgabe 13.2 (Vektoren; Datei: `vectors.py`; 4 Punkte)

In dieser Aufgabe sollen Ihre Funktionsdefinitionen, außer einer `return`-Anweisung, keine weiteren Anweisungen enthalten, oder wie im Hinweis als Variable mit Funktionswert definiert werden.

Die folgende Datenklasse beschreibt 3-dimensionale Vektoren:

```
@dataclass
class V3:
    x: int
    y: int
    z: int
```

Die komponentenweise Addition zweier Vektoren ist definiert durch:

```
cadd = lambda v, w: V3(v.x + w.x, v.y + w.y, v.z + w.z)
```

Die komponentenweise Multiplikation zweier Vektoren ist definiert durch:

```
cmul = lambda v, w: V3(v.x * w.x, v.y * w.y, v.z * w.z)
```

Schreiben Sie eine Funktion `mapV3`, die eine zweistellige[1] Funktion `f` auf ganzen Zahlen als Argument nimmt und eine zweistellige Funktion auf Vektoren zurückgibt, welche `f` komponentenweise auf den Vektoren anwendet.

Verwenden Sie `mapV3` um die komponentenweise Addition, Multiplikation, Subtraktion, ganzzahlige Division und Exponentation durch jeweils einen einzigen `mapV3`-Aufruf zu definieren:

```
cadd = mapV3(...)
cmul = mapV3(...)
csub = mapV3(...)
cdiv = mapV3(...)
cpow = mapV3(...)
```

[1] Eine zweistellige Funktion ist eine Funktion, die zwei Argumente entgegen nimmt, wie z.B. Addition.

Aufgabe 13.3 (Comprehensions; Datei: `comprehensions.py`; Punkte: 2+3)

In dieser Aufgabe sollen Ihre Funktionsdefinitionen, außer einer `return`-Anweisung, keine weiteren Anweisungen enthalten, oder wie im Hinweis als Variable mit Funktionswert definiert werden.

(a) (2 Punkte) Implementieren Sie die Funktion `normalize_word` aus Aufgabe 9.1.(b) erneut, aber verwenden Sie hierzu die `String.join`-Methode und eine Generator-Comprehension:

> Schreiben Sie eine Funktion `normalize_word`, die einen String als Argument nimmt, alle Buchstaben in Kleinbuchstaben umwandelt, alle Zeichen entfernt, die keine Buchstaben sind, und den resultierenden String zurückgibt. Beispiel.
>
> ```
> >>> normalize_word("Ver-BieTen?")
> "verbieten"
> ```
>
> Hinweis: Es bietet sich an die `str`-Methoden `isalpha` und `lower` zu verwenden.

(b) (3 Punkte) Eine Liste `xs` ist sortiert, wenn für alle Listen-Indexe `i` und `j` gilt, dass wenn `i <= j` dann `xs[i] <= xs[j]`.

Schreiben Sie eine Funktion `is_sorted`, welche nach diesem Verfahren prüft ob eine Liste sortiert ist. Verwenden Sie dafür eine Kombination aus der `all`-Funktion und einer Generator-Comprehension.

Hinweis: Die `all`-Funktion verknüpft alle Elemente eines `bool`-Iterable mit der `and` Funktion, z.B. `all([x, y, z]) == (x and y and z)`.

Aufgabe 13.4 (Reducing Bits; Datei: `bits_to_int.py`; Punkte: 3)

In dieser Aufgabe sollen Ihre Funktionsdefinitionen, außer einer `return`-Anweisung, keine weiteren Anweisungen enthalten, oder wie im Hinweis als Variable mit Funktionswert definiert werden.

Schreiben Sie eine Funktion `bits_to_int`, die eine Liste von Bits als Argument nimmt und die zugehörige postive ganze Zahl zurückgibt.

Die Bits sollen im *Big-Endian*-Format interpretiert werden, d.h. das erste Bit hat wie gewohnt den größten Exponenten. Beispiel:

```
assert bits_to_int([1,0,1,0]) == 10
```

Verwenden Sie hierfür die `reduce`-Funktion aus dem Modul `functools`.

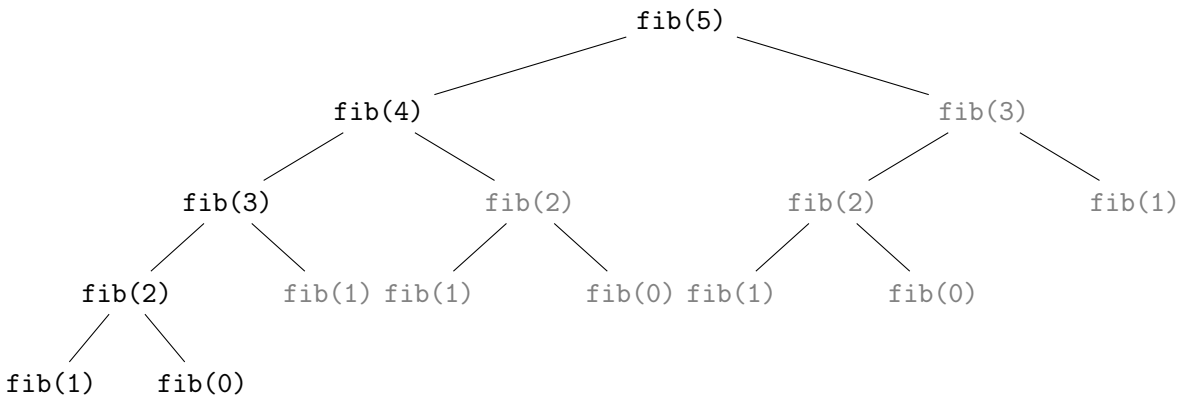

Abbildung 1: Rekursive Funktionsaufrufe für `fib(5)`

Aufgabe 13.5 (Magische Dekoratoren; Datei: `decorators.py`; Punkte: 4 (**schwer**))
Die Fibonacci-Folge kann rekursiv über folgende Funktion beschrieben werden:

```
def fib(n: int) -> int:
    if n == 0:
        return 0
    elif n == 1:
        return 1
    else:
        return fib(n-1) + fib(n-2)
```

Auch wenn diese Funktion auf den ersten Blick eher unschuldig aussieht, hat sie es doch ganz schön in sich: um `fib(n)` zu berechnen, muss ungefähr $2^{\texttt{n}}$ mal die `fib`-Funktion ausgewertet werden. Die Ausführungszeit von `fib` wächst also exponentiell mit der Größe von `n`. Auf modernen Computern ist die Ausführung für `n<20` blitzschnell, für `n=30` dauert es bereits Sekunden und für `n>40` eine Ewigkeit.

Die Fibonacci-Funktion kann jedoch auch sehr viel effizienter implementiert werden. Der Trick basiert dabei auf der Beobachtung, dass viele der rekursiven Aufrufe doppelt ausgeführt werden. In Abbildung 1 sieht man die rekursiven Funktionsaufrufe, die für `fib(5)` nötig sind, als Baum visualisiert. Die Knoten von doppelten Funktionsaufrufe sind in grauem Text hervorgehoben. Würden wir uns z.B. das Ergebnis von `fib(3)` im linken Teilbaum merken, dann könnten wir es auf der rechten Seite einfach nachschlagen, und der gesamte rechte Teilbaum von `fib(3)` würde wegfallen. Macht man dies für alle `n`, so fallen alle grauen Knoten weg, und es bleibt (fast) eine Linie von `n` Knoten übrig - wir können `fib(n)` also mit nur `n` rekursiven Aufrufen berechnen.

Der folgende Code implementiert solch eine optimierte Fibonacci-Funktion. Hierbei wird ein Dictionary `cache` verwendet, in welchem wir uns die Argumente und Rückgabewerte der bisherigen Funktionsaufrufe merken.

```
def fib_fast(n: int) -> int:
    return fib_fast_cache(n, dict())

def fib_fast_cache(n: int, cache: dict[int,int]) -> int:
    # If we already computed fib(n), then return the previously computed result.
    if n in cache:
        return cache[n]

    # Otherwise we compute the result,
    result = None
    if n == 0:
        result = 0
    elif n == 1:
        result = 1
    else:
        result = fib_fast_cache(n-1, cache) + fib_fast_cache(n-2, cache)

    # put the result in the cache - in case we need it again later,
    cache[n] = result

    # and return the result.
    return result
```

Ihre Aufgabe ist es nun einen Decorator `cached` zu implementieren, welcher es erlaubt den Code von `fib` hinzuschreiben, aber die optimierte Implementierung von `fib_fast` zu erhalten:

```
@cached
def fib_fast_and_simple(n: int) -> int:
    if n == 0:
        return 0
    elif n == 1:
        return 1
    else:
        return fib_fast_and_simple(n-1) + fib_fast_and_simple(n-2)
```

`cached(f)` soll also beliebige einstellige Funktionen `f` so dekorieren, dass ihre Funktionsaufrufe in einem Dictionary zwischengespeichert und bei Bedarf wieder abgerufen werden.

Verwenden Sie wie in der Vorlesung die `time.time()`-Funktion, um die Ausführungszeiten der Funktionsaufrufe zu vergleichen. Auf meinem Rechner benötigt der Aufruf `fib(32)` in etwa 0.83 Sekunden um den Wert 2178309 zu berechnen, wohingegen `fib_fast` und `fib_fast_and_simple` lediglich 0.00011 Sekunden benötigen. Je größer `n` ist, desto drastischer wird der Unterschied.

Hinweis: Die `wrapper`-Funktion des Dekorators muss sich ein Dictionary merken, welches mehrere Funktionsaufrufe überlebt. Dies erreicht man durch das Einfangen

einer Variable, die man außerhalb des `wrapper`s definiert (variable capture).

Hinweis: Bei einer rekursiven Funktion wirkt sich ein Dekorator auch auf die rekursiven Aufrufe auf.

Aufgabe 13.6 (Erfahrungen; 2 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.