Einführung in die Programmierung

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

Universität Freiburg
Institut für Informatik
Wintersemester 2023

Übungsblatt 13

Abgabe: Montag, 29.01.2024, 9:00 Uhr morgens

Aufgabe 13.1 (Magische Dekoratoren; Datei: `decorators.py`; Punkte: 7)

In dieser Aufgabe müssen Sie *keine* Typannotationen schreiben.
Die Fibonacci-Folge kann rekursiv über folgende Funktion beschrieben werden:

```python
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^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<25` blitzschnell, für `n=35` dauert es bereits Sekunden und für `n>45` eine Ewigkeit.

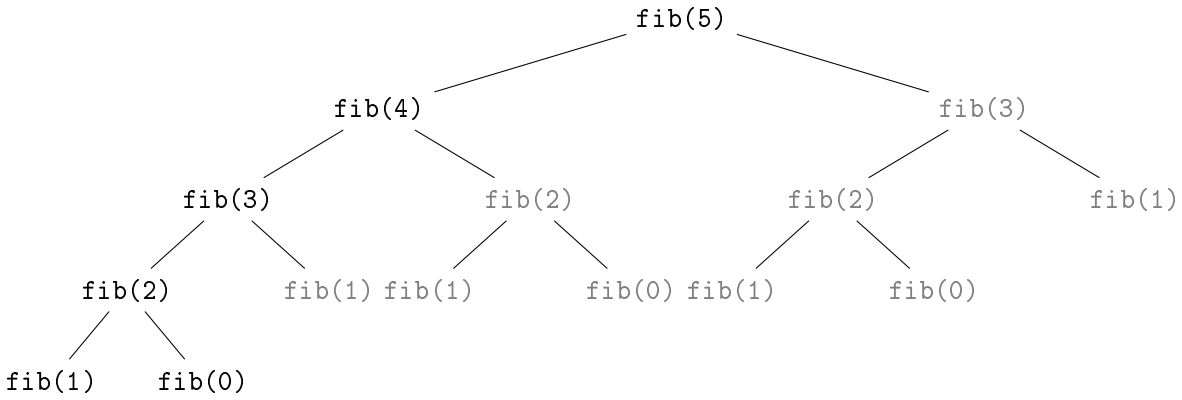


Abbildung 1: Rekursive Funktionsaufrufe für `fib(5)`

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 Funktionsaufrufen 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:
    cache: dict[int, int] = dict()

    def fib_fast_cache(n: 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) + fib_fast_cache(n - 2)
        # put the result in the cache - in case we need it again later,
        cache[n] = result
        # and return the result.
        return result
    return fib_fast_cache(n)
```

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 Ausfüh-

rungszeiten der Funktionsaufrufe zu vergleichen. Auf meinem Rechner benötigt der Aufruf `fib(32)` in etwa 0.27 Sekunden um den Wert 2178309 zu berechnen, wohingegen `fib_fast` und `fib_fast_and_simple` lediglich 0.00004 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 13.2 (Closures; 7 Punkte; Datei: `closures.py`)

In dieser Aufgabe müssen Sie *keine* Typannotationen schreiben.

```
def cons(*args):
    def tup(f):
        return f(*args)
    return tup
```

Die Funktion `cons` nimmt eine beliebige Anzahl von Argumenten `*args` und gibt eine Closure `tup` zurück, die das Tupel `args` als freie Variable hat.

(a) `car`; **1.5 Punkte**
Schreiben Sie die Funktion `car`, die eine von `cons` zurückgegebene Closure `t` als Argument nimmt und den ersten Wert des Tupels von `t` zurückgibt. Sie dürfen davon ausgehen, dass das Tupel mindestens einen Wert besitzt.

```
example1 = cons(3, 5, 1, 4)
example2 = cons(1)
assert car(example1) == 3
assert car(example2) == 1
```

(b) `cdr`; **1.5 Punkte**
Schreiben Sie die Funktion `cdr`, die eine von `cons` zurückgegebene Closure `t` als Argument nimmt und den letzten Wert des Tupels von `t` zurückgibt. Sie dürfen davon ausgehen, dass das Tupel mindestens einen Wert besitzt.

```
assert cdr(example1) == 4
assert cdr(example2) == 1
```

(c) `idx`; **1.5 Punkte**
Schreiben Sie die Funktion `idx`, die eine von `cons` zurückgegebene Closure `t` und eine Ganzzahl `n` als Argumente nimmt und den `n`-ten Wert des Tupels von `t` zurückgibt. Falls `n` kein valider Index ist, soll `None` zurückgegeben werden.

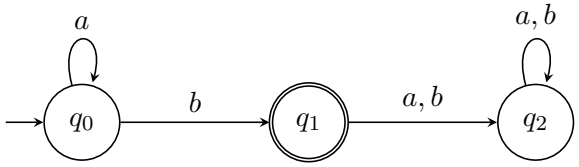
```
assert idx(example1, 2) == 1
assert idx(example1, -3) is None
assert idx(example1, 7) is None
```

(d) `counter`; **2.5 Punkte**
Schreiben Sie eine Funktion `counter`, die eine Closure (ohne Parameter) zurückgibt, die als Zähler dient und bei jedem Aufruf die jeweils nächste natürliche Zahl zurückgibt (beginnend bei 0).

```
c1 = counter()
assert c1() == 0
assert c1() == 1
c2 = counter()
assert (c2(), c2(), c2()) == (0, 1, 2)
assert c1() == 2
assert c1() == 3
```

Aufgabe 13.3 (Automaton; 6 Punkte; Datei: `automaton.py`)

In der Datei `automaton.py` befindet sich die in der Vorlesung vorgestellte Definition der `Automaton`-Klasse. In dieser erstellen sie lediglich eine Instanz dieser Klasse, die den folgenden Automaten implementiert:



(a) `State`; **1.5 Punkte**
Schreiben Sie ein `Enum` `State`, das alle Knoten des Automaten repräsentiert.

(b) `delta` **3 Punkte**
Schreiben Sie eine Funktion `delta`, die einen State `state` und einen String `input` nimmt und den Folgezustand zurückgibt. Verwenden Sie hierzu Pattern-Matching wie in der Vorlesung gezeigt. Der jeweilige Folgezustand ist durch das Diagram oben gegeben.

(c) `test_automaton` **1.5 Punkte**
Schreiben Sie eine Funktion `test_automaton` ohne Argumente, in der Sie nun eine Instanz des besagten Automats erstellen, und schreiben Sie zudem `assert`-Statements, die diesen mit ein paar Eingaben testen (mindestens ein `True` und `False` Beispiel).

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.