Einführung in die Programmierung

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

Universität Freiburg
Institut für Informatik
Wintersemester 2024

Übungsblatt 13

Abgabe: Montag, 27.01.2025, 9:00 Uhr

Aufgabe 13.1 (Closures; 7 Punkte; Datei: `closures.py`)

Closures können die Variablen, die bei ihrer Definition verfügbar sind, einfangen. Das heißt, selbst wenn die umgebende Funktion bereits beendet ist, können Closures noch auf diese Variablen zugreifen. Ein typischer Verwendungszweck sind *Funktionsfabriken*, die angepasste Funktionen zurückgeben können. Funktionen aus einer Funktionsfabriken können zum Beispiel bestimmte Werte zugreifen können, die während der Erstellung der Closure verfügbar waren.

(a) `multiply`; **1.5 Punkte**
Schreiben Sie die Funktion `multiply`, die eine Ganzzahl `factor` als Argument nimmt und eine Closure zurückgibt, die eine weitere Ganzzahl `x` als Argument nimmt und das Produkt von `factor` und `x` zurückgibt.

```
>>> mult_2 = multiply(2)
>>> mult_2(3)
6
>>> mult_5 = multiply(5)
>>> mult_5(3)
15
```

(b) `create_logger`; **1.5 Punkte**
Man kann Closures auch dafür verwenden, Zustände zu speichern. Ein gutes Beispiel ist die Erstellung von Loggern. Logger werden in der Regel verwendet, um Informationen über den Verlauf eines Programms zu speichern. Dabei wird zwischen verschiedenen Log-Levels wie Debug, Info, Warn und Error unterschieden. Erstellen Sie zuerst eine Funktion `create_logger`, die einen Präfix `prefix` als Argument nimmt und eine Closure zurückgibt, die eine Nachricht `msg` als Argument nimmt und die Nachricht `prefix + msg` zurück gibt. Außerdem soll die Closure alle bisher ausgegebenen Nachrichten choronologisch in einer Liste speichern und diese zurückgeben, wenn sie ohne Argumente aufgerufen wird.

```
>>> debug_logger = create_logger("DEBUG: ")
>>> debug_logger("Hello")
'DEBUG: Hello'
>>> debug_logger("World")
'DEBUG: World'
>>> error_logger = create_logger("ERROR: ")
```

```
>>> error_logger("Whatson, we have a problem")
'ERROR: Whatson, we have a problem'
>>> error_logger()
['ERROR: Whatson, we have a problem']
>>> debug_logger()
['DEBUG: Hello', 'DEBUG: World']
```

(c) `log_manager`; **1.5 Punkte**

Der derzeitige Logger speichert die Logs von unterschiedlichen Levels in getrennten Listen. Das wollen wir ändern. Erstellen Sie eine Funktion `log_manager`, die eine zentrale Liste von Logs speichert und eine Funktion zurückgibt, die ähnlich wie `create_logger` Logger erstellt, die aber alle in die zentrale Liste schreiben. Die Logs sollen außerdem nicht mehr ausgegeben werden, wenn der Logger ohne Argumente aufgerufen wird, stattdessen soll `log_manager` eine weitere Funktion zurückgeben, die beim Aufruf die gespeicherten Logs zurückgibt.

```
>>> create_logger, get_logs = log_manager()
>>> debug_logger = create_logger("DEBUG: ")
>>> debug_logger("Hello")
'DEBUG: Hello'
>>> debug_logger("World")
'DEBUG: World'
>>> error_logger = create_logger("ERROR: ")
>>> error_logger("Houston, we have a problem")
'ERROR: Houston, we have a problem'
>>> get_logs()
['DEBUG: Hello', 'DEBUG: World', 'ERROR: Houston, we have a problem']
```

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

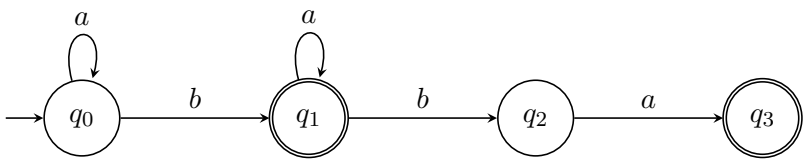
Aufgabe 13.2 (Automaten; 6 Punkte; Datei: `automaton.py`)

In der Vorlesung haben Sie Deterministische Endliche Automaten kennengelernt:

```python
@dataclass(frozen=True)
class Automaton[Q, E]:  # Zustände und Eingabealphabet
    delta: Callable[[Q, E], Q]  # Transitionsfunktion
    start: Q  # Startzustand q0
    finals: frozenset[Q]  # Menge von Endzuständen F

    def accept(self, input: Iterable[E]) -> bool:
        state = self.start
        for c in input:
            state = self.delta(state, c)
        return state in self.finals
```

In dieser Aufgabe erstellen wir eine Instanz der Klasse `Automaton`, die den folgenden Automaten implementiert:



Wie in der Vorlesung erläutert, nehmen wir an, dass alle nicht gezeigten Kanten in einen absorbierenden Fehlerzustand `qe` laufen. Beachten Sie dies in den folgenden Aufgaben.

(a) `State` und `Alphabet` **(1.5 Punkte)**
Implementieren Sie ein `Enum State`, das alle Zustände des Automaten repräsentiert. Benennen Sie die Felder des `Enum`s ausschließlich mit Großbuchstaben und Zahlen[1]. Zum Beispiel soll der Zustand `q0` durch das `Enum`-Feld `State.Q0` dargestellt werden. Die Werte der Felder können Sie frei wählen.

Definieren Sie außerdem einen Literal-Typen `Alphabet`, der alle Symbole des Eingabealphabets beschreibt.

(b) `delta` **(3 Punkte)**
Implementieren Sie eine Funktion `delta`, die einen Zustand `state` und ein Eingabesymbol `input` als Parameter erhält und den Folgezustand zurückgibt. Verwenden Sie dabei Pattern Matching, wie es in der Vorlesung gezeigt wurde. Die möglichen Übergänge können Sie dem oben gezeigten Diagramm entnehmen.

(c) `test_automaton` **(1.5 Punkte)**
Schreiben Sie eine Funktion `test_automaton`, die keine Argumente entgegennimmt. In dieser Funktion soll eine Instanz des oben beschriebenen Automaten

[1] Das ist eine gängige Konvention

erstellt werden. Fügen Sie außerdem `assert`-Statements hinzu, um den Automaten mit verschiedenen Eingaben zu testen (mindestens ein Beispiel, das `True` zurückgibt, und eines, das `False` ergibt).

Aufgabe 13.3 (Magische Dekoratoren; Datei: `decorators.py`; Punkte: 7)

In Aufgabenblatt 11 haben Sie bereits die Fibonacci-Folge kennengelernt und einen nicht-rekursiven Generator dafür implementiert. Die Fibonacci-Folge kann auch rekursiv über folgende Funktion beschrieben werden:

```
def fib(n: int) -> int:
    match n:
        case 0:
            return 0
        case 1:
            return 1
        case _:
            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.

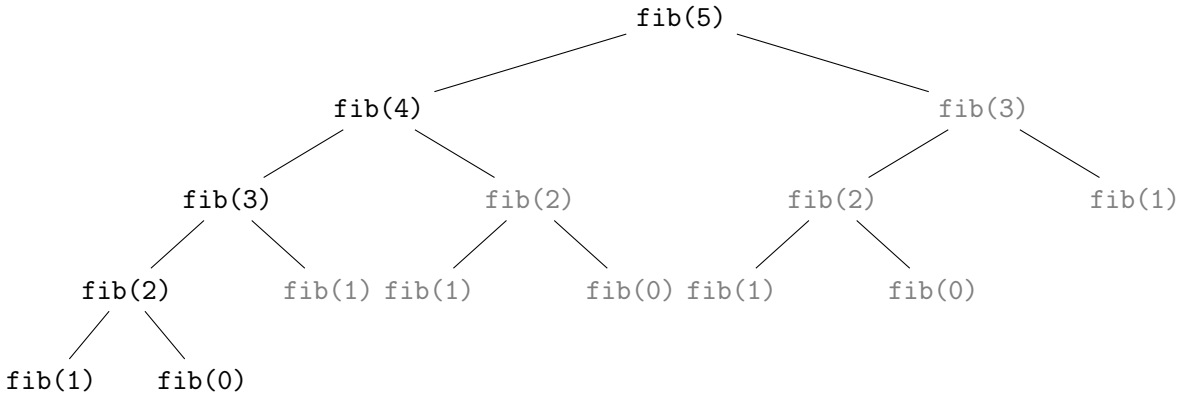


Abbildung 1: Rekursive Funktionsaufrufe für `fib(5)`

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:
        # Falls fib(n) schon berechnet wurde, gebe den Wert zurück
        if n in cache:
            return cache[n]
        # Andernfalls berechne den Wert
        result = None
        match n:
            case 0:
                result = 0
            case 1:
                result = 1
            case _:
                result = fib_fast_cache(n - 1) + fib_fast_cache(n - 2)
        # Speichere den neu berechneten Wert
        cache[n] = 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:
    match n:
        case 0:
            return 0
        case 1:
            return 1
        case _:
            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.

Schreiben Sie nun eine Funktion `compare`, die ein Argument `n` entgegennimmt und die Ausführungszeiten von `fib` und `fib_fast_and_simple` angewendet auf `n` misst

und diese im Terminal auf 6 Nachkommastellen gerundet *ausgibt* (`print`). Verwenden Sie dazu die `time()`-Funktion aus dem gleichnamigen Modul `time`.

Die Ausgabe kann beispielsweise so aussehen[2]:

```
>>> compare(12)
fib(12)=144 took 0.000033s
fib_fast_and_simple(12)=144 took 0.000007s
>>> compare(35)
fib(35)=9227465 took 2.554888s
fib_fast_and_simple(35)=9227465 took 0.000060s
```

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 13.4 (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 Webplattform. 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.

[2] Abhängig von Ihren Rechner, können die Ausführungszeiten natürlich variieren. Für größere Werte n sollte jedoch `fib_fast_and_simple` deutlich schneller sein, als `fib`.