Informatik I: Einführung in die Programmierung

Prof. Dr. Peter Thiemann
Hannes Saffrich, Michael Uhl
Wintersemester 2022

Universität Freiburg
Institut für Informatik

Übungsblatt 13

Abgabe: Montag, 30.01.2023, 9:00 Uhr morgens

Hinweis: Es gelten die selben Regeln wie bisher, diese können in Blatt 10 eingesehen werden.

Aufgabe 13.1 (Exceptions; `suppress.py`; Punkte: 4)

Schreiben Sie eine Funktion `suppress(f: Callable, ignore: tuple)`, welche eine parameterlose Funktion `f` und ein Tuple an `Exception`s `ignore` als Argumente erhält und eine neue Funktion `g` zurückgibt. `g` soll sich dabei identisch zu `f` verhalten, solange beim Aufruf keine `Exception` aus `ignore` auftritt. Tritt während des Aufrufs `f()` eine solche `Exception` auf, so soll diese beim Aufruf `g()` ignoriert und `None` zurückgeben werden. Beispiel:

```
>>> from functools import partial
>>> from typing import Callable
>>> def foo(n: int) -> int:
...     return 35 // n
...
>>> assert suppress(partial(foo, 1), ())() == 35 == foo(1)
>>> suppress(partial(foo, 0), (ZeroDivisionError,))()
>>> suppress(partial(foo, 0), ())()
Traceback (most recent call last):
  ...
  File "suppress.py", line 26, in <lambda>
    suppress(lambda: 3 / 0, ())()
ZeroDivisionError: division by zero
```

Verwenden Sie zum Lösen dieser Aufgabe verschachtelte Funktionen.

Aufgabe 13.2 (State Machines; Datei: `state_machine.py`; Punkte: 2+2+4)

In der Vorlesung haben Sie State Machines kennengelernt und Sie haben nun die Idee, ein paar Funktionen Ihres Smartphones als State Machine abzubilden.

Erstellen Sie, wie in der Vorlesung gezeigt, mit Hilfe von `enum.Enum` sowie `enum.auto` einen Zustandsautomaten `MyState` und die dazugehörige Funktion `next_state()`. Die Funktion erhält einen Zustand `state` und eine Eingabe `input` (`str`) und gibt wiederum einen Zustand `MyState` zurück.

Dieser Automat erhält in allen Aufgabenteilen Sensordaten vom Typ `str` und verhält sich wie folgt:

(a) Zuerst befindet sich der Automat im Zustand `INIT`. Sendet der Beschleunigungssensor zweimal hintereinander das Signal `"SENS_ACC"`, so soll das eingebaute Licht eingeschaltet werden, was dem Zustand `LIGHT_ON` entspricht. Sendet der Beschleunigungssensor nun erneut zweimal hintereinander sein Signal, so befindet sich das Smartphone wieder im Ausgangszustand. Ungültige Signale werden hier ignoriert und es wird der aktuelle Zustand zurückgegeben.

(b) Weiter gibt es einen Berührungssensor, der ein Antippen des Displays erkennt und das Signal `"SENS_TAP"` sendet. Zweimaliges Antippen direkt hintereinander führt zu dem Zustand `SCREEN_ON` und erneutes zweimaliges Antippen führt wieder zum Ausgangszustand. Beide Zustände lassen sich zudem zu einem weiteren Zustand `SCREEN_LIGHT_ON` kombinieren, bei dem sowohl Licht als auch Display eingeschaltet sind. Werden erneut zwei dieser Signale hintereinander wahrgenommen, so schaltet sich das Licht, respektive der Bildschirm aus. Ungültige Signalfolgen führen zum letzten gültigen Zustand (`INIT`, `LIGHT_ON`, `SCREEN_ON` oder `SCREEN_LIGHT_ON`). Ungültige Signale werden wieder ignoriert.

(c) Sie bemerken, dass es sehr schnell unübersichtlich werden kann, wenn Sie diesem Schema weitere Zustände hinzufügen wollen. Sie entsinnen sich, dass in der Vorlesung hierfür generische Klassen verwendet wurden. Implementieren Sie obige State Machine mit Hilfe von generischen Klassen, wie es in der Vorlesung gezeigt wurde und benennen Sie nun die Klassen exakt wie in der Vorlesung (`State`, `S_After`, etc). Die `TypeVar`-Statements dürfen Sie ebenso wie `S = MyState` ausnahmsweise in den top-level-Bereich schreiben.

Verwenden Sie für die `output()`-Methode weiterhin die vorher definierten Zustände. Verwenden Sie weiter die in der Vorlesung definierte `automaton`-Funktion, um ihre Zustandsmaschine zu testen. Ersetzen Sie hierfür das `print`- mit einem entsprechenden `yield`-Statement, was zudem ein einziges Mal vor der Schleife aufgerufen werden muss, damit alle Elemente ausgegeben werden. Sie müssen keine Tests hierzu schreiben, es wird ihnen jedoch empfohlen.

Nachtrag: gemeint ist die folgende `automaton`-Funktion:

```
def automaton(input: Iterable[str]):
    state: State[S | str, S] = S_Init()

    for x in input:
        state = state.next(x)
        print(state.output())
```

Aufgabe 13.3 (CSV Parsing; Datei: `csv_parser.py`; Punkte: 2+2+2+0)

Das CSV-Format (Comma Separated Values) erlaubt es Tabellen in Textdateien darzustellen, sodass diese einfach maschinell verarbeitet werden können.

Jede Zeile einer `.csv`-Datei entspricht dabei der Zeile einer Tabelle. Innerhalb einer Zeile werden die Tabellenzellen durch Kommas getrennt.[1]

CSV findet zum Beispiel beim Online-Banking Anwendung. Die meisten Anbieter erlauben es z.B. die Umsatzübersicht als `umsatz.csv` mit folgendem Inhalt zu exportieren:

```
Datum,Verwendungszweck,Betrag
30.12.2020,Bafoeg-Foerdergeld,+514.00
01.01.2021,Miete,-400.00
03.01.2021,Bestellung im Amazonas,-43.20
```

In dieser Aufgabe sollen Ihre Generator-Funktionen nur die Berechnungen durchführen, die für das jeweils nächste Element benötigt werden.

Das Beispiel `umsatz.csv` finden Sie auch auf der Vorlesungsseite.

(a) Schreiben Sie eine Generator-Funktion `lines`, die den Dateipfad einer Textdatei als Argument nimmt und deren Zeilen generiert. Beispiel:

```
>>> for line in lines("umsatz.csv")
>>>     print(line)
Datum,Verwendungszweck,Betrag
30.12.2020,Bafoeg-Foerdergeld,+514.00
01.01.2021,Miete,-400.00
03.01.2021,Bestellung im Amazonas,-43.20
```

(b) Schreiben Sie eine Generator-Funktion `parse_csv`, die einen Generator von Strings als Argument nimmt, diese als Zeilen einer CSV-Datei auffasst und die zugehörigen Listen von Zellen generiert. Beispiel:

```
>>> list(parse_csv(["Datum,Verwendungszweck,Betrag",
                    "30.12.2020,Bafoeg-Foerdergeld,+514.00"]))
[ ["Datum", "Verwendungszweck", "Betrag"],
  ["30.12.2020", "Bafoeg-Foerdergeld", "+514.00"] ]
```

[1] Das `CSV`-Format erlaubt es eigentlich auch Strings zu verwenden, sodass Kommas innerhalb einer Tabellenzelle verwendet werden können. Zum Beispiel beschreibt dann `"[23,4]",25` zwei Spalten statt drei. Sie können dieses Feature in dieser Aufgabe ignorieren.

Verwenden Sie hierbei *nicht* das `csv`-Modul aus der Standard-Bibliothek.

(c) Schreiben Sie eine Funktion `update_balance`, die als Argument einen Startkontostand `balance` und einen Dateipfad `csv_path` nimmt, und den Kontostand nach dem Ausführen der Transaktionen zurückgibt. Verwenden Sie hierzu die Funktionen `lines` und `parse_csv` aus den vorherigen Aufgabenteilen. Sie können dabei annehmen, dass `csv_path` eine gültige `.csv`-Datei beschreibt, die eine Tabelle mit dem gleichen Format wie im Beispiel enthält. Beispiel:

```
>>> update_balance(100.00, "umsatz.csv")
170.8   # == 100.0 + 514.0 - 400.0 - 43.2
```

(d) Atmen Sie tief durch und freuen Sie sich, dass ihr Code auch in der Lage wäre den Umsatz von Amazon zu verarbeiten. Hier könnte die `umsatz.csv` mehrere Terrabyte groß sein, aber Ihr Arbeitsspeicher ist im Regelfall nur ein paar Gigabyte groß. Würde Ihr Programm zunächst versuchen die gesamte Datei / Tabelle in den Speicher zu laden und dann erst den neuen Kontostand berechnen, dann könnten Sie diese Dateigrößen nicht verarbeiten, da Ihnen der Arbeitsspeicher ausgehen würde. Die Generatoren erlauben es dabei die einzelnen Verarbeitungsschritte wie gewohnt in verschiedene Funktionen zu unterteilen - ein weiterer Baustein um selbst komplexe Programme noch einigermaßen verständlich und wiederverwendbar aufzuschreiben.

Aufgabe 13.4 (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.