Einführung in die Programmierung

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

Universität Freiburg
Institut für Informatik
Wintersemester 2024

Übungsblatt 7

Abgabe: Montag, 02.12.2024, 9:00 Uhr

Aufgabe 7.1 (Maschinensteuerung; 7 Punkte; Datei: `simulation.py`)

In dieser Aufgabe sollen Sie eine Simulation für Maschinen in einer Fabrik mit verschiedenen Zuständen und Aktionen implementieren. Verwenden Sie zum Lösen dieser Aufgabe Pattern-Matching und **keine** `if`-Anweisungen.

Gehen Sie dabei wie folgt vor:

(a) (1 Punkt) Definieren Sie einen *Aufzählungstyp* `State` mit den Feldern `RUNNING` (Maschine läuft) und `IDLE` (Maschine ist im Stillstand), die die möglichen Zustände der Maschine repräsentieren.

Definieren Sie eine unveränderliche[1] (`frozen=True`) Datenklasse `Machine` mit einem Attribut `state` vom Typ `State`, das den aktuellen Zustand der Maschine speichert und einem Attribut `time_left` (Ganzzahl), das angibt, wie viele Zeiteinheiten die Maschine noch laufen soll.

(b) (2 Punkte) Schreiben Sie eine Funktion `time_step`, die eine Maschine als Argument erhält und die Maschine um eine Zeiteinheit weiterführt. Wenn die Maschine gerade läuft, soll die Funktion die Zeit, die die Maschine noch laufen soll, um eins verringern. Wenn die Maschine in diesem Zeitschritt fertig wird (`time_left == 0`), soll sie in den Zustand `IDLE` wechseln. Ist die Maschine im Zustand `IDLE`, soll sie unverändert bleiben.

Sie können annehmen, dass Maschinen nie negative Restlaufzeiten haben und dass Maschinen, die laufen, immer mindestens eine Zeiteinheit Restlaufzeit haben.

```
>>> time_step(Machine(State.RUNNING, 42))
Machine(state=<State.RUNNING: 'running'>, time_left=41)
>>> time_step(Machine(State.RUNNING, 1))
Machine(state=<State.IDLE: 'idle'>, time_left=0)
>>> time_step(Machine(State.IDLE, 1337))
Machine(state=<State.IDLE: 'idle'>, time_left=1337)
>>> time_step(Machine(State.IDLE, 0))
Machine(state=<State.IDLE: 'idle'>, time_left=0)
```

[1] Um den Lesefluss zu verbessern, verwenden wir in dieser Aufgabe die Formulierung *eine Maschine verändern* und meinen damit, dass Sie eine *neue* Maschine zurückgeben sollen, die entsprechend abgeändert ist.

(c) (2 Punkte) Definieren Sie eine *Enumeration* `Action` mit den Feldern `START` (Maschine wird gestartet), `STOP` (Maschine wird gestoppt) und `NONE` (keine Aktion).

Schreiben Sie eine Funktion `apply_action`, die eine Maschine und eine Aktion als Argumente erhält und die Maschine entsprechend der Aktion verändert. Wird die Maschine gestartet, soll ihr Zustand auf `State.RUNNING` sein, wird sie gestoppt auf `State.IDLE`. Wird keine Aktion ausgeführt, so soll die Maschine unverändert bleiben. Maschinen, die keine Restlaufzeit mehr haben, sollen nicht gestartet werden können. Beachten Sie, dass Aktionen keine Zeit zum Ausführen benötigen!

```
>>> apply_action(Machine(State.IDLE, 42), Action.START)
Machine(state=<State.RUNNING: 'running'>, time_left=42)
>>> apply_action(Machine(State.IDLE, 0), Action.START)
Machine(state=<State.IDLE: 'idle'>, time_left=0)
>>> apply_action(Machine(State.RUNNING, 42), Action.STOP)
Machine(state=<State.IDLE: 'idle'>, time_left=42)
>>> apply_action(Machine(State.RUNNING, 0), Action.STOP)
Machine(state=<State.IDLE: 'idle'>, time_left=0)
```

(d) (2 Punkte) Schreiben Sie eine Funktion `simulate`, die eine Maschine `machine` und eine Liste von Aktionen `actions` als Argumente erhält. Die Funktion soll die Maschine für `len(actions)` Zeitschritte simulieren und den Verlauf der Maschine als Liste (`list[Machine]`) zurückgeben. In jedem Zeitschritt wird zunächst die jeweilige Aktion aus `actions` ausgeführt und dann die resultierende Maschine für einen Zeitschritt simuliert. Verwenden Sie hierzu die Funktionen `apply_action` und `time_step`.

```
>>> machine = Machine(State.IDLE, 7)
>>> simulate(machine, [])
[Machine(state=<State.IDLE: 'idle'>, time_left=7)]
>>> simulate(machine, [Action.NONE])
[Machine(state=<State.IDLE: 'idle'>, time_left=7),
↪  Machine(state=<State.IDLE: 'idle'>, time_left=7)]
>>> act = [Action.START, Action.NONE, Action.STOP, Action.START]
>>> simulate(machine, act)
[Machine(state=<State.IDLE: 'idle'>, time_left=7),
↪  Machine(state=<State.RUNNING: 'running'>, time_left=6),
↪  Machine(state=<State.RUNNING: 'running'>, time_left=5),
↪  Machine(state=<State.IDLE: 'idle'>, time_left=5),
↪  Machine(state=<State.RUNNING: 'running'>, time_left=4)]
```

Hinweis: In den vorherigen Code-Beispielen wurde `State.RUNNING` auf den Wert `"running"` und `State.IDLE` auf den Wert `"idle"` abgebildet. Wenn Sie in Aufgabenteil (a) andere Werte für das Enum `State` verwendet haben, sehen Sie diese in der Ausgabe des interaktiven Modus!

Aufgabe 7.2 (Bäume; 3 Punkte; Datei: `trees.py`)

In der Vorlesung haben Sie Bäume als Datenstruktur kennengelernt:

```
@dataclass
class Node[T]:
    mark : T
    left : Optional['Node[T]'] = None
    right: Optional['Node[T]'] = None


type BTree[T] = Optional[Node[T]]
```

Schreiben Sie eine Funktion `tree_max`, die einen binären Baum `btree` als Argument nimmt und das Maximum aller Werte in `btree` zurückgibt. Falls `btree` leer ist, soll `None` zurückgegeben werden. **Verwenden Sie Rekursion und Pattern Matching.**

Sie können davon ausgehen, dass `max_tree` einen `BTree` nimmt, dessen Blätter Werte vom Typ `int` oder `str` enthalten. Beachten Sie dies auch in der Typannotation der Funktion.

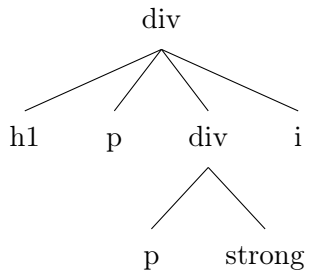
```
>>> tree_max(None)  # None
>>> tree_max(Node(3))
3
>>> tree_max(Node("Baum"))
'Baum'
>>> tree_max(Node("Hallo", Node("Welt"), Node(":)")))
'Welt'
>>> tree_max(Node(18, Node(5, Node(19)), Node(77, Node(-3), Node(9))))
77
```

Aufgabe 7.3 (HTML Modellieren; Punkte: 10; Datei: `mini_html.py`)

HTML steht für *Hypertext Markup Language* und nutzt Tags, um bestimmte Elemente eines Dokuments zu verändern. Ähnlich wie bei einer einfachen Textdatei (z.B. im `.txt`-Format) kann auch in HTML-Dateien normaler Text geschrieben werden. Um dem Dokument Struktur zu geben, werden Tags der Form `<tag> Text </tag>` benutzt. Der einfachste Tag ist `<p>` und beinhaltet normalen Text. Außerdem gibt es die Tags `<strong>` und `<i>` mit denen Text **dick** oder *kursiv* dargestellt werden kann. Überschriften werden mit `<h1>` gekennzeichnet. Darüber hinaus dienen Tags nicht nur der Textformatierung, sondern auch der Strukturierung des Dokuments. Ein gängiges Element zur Strukturierung ist das `<div>`-Tag, welches als Container für andere Elemente dient. Da Tags ineinander geschachtelt werden können, lässt sich die Struktur einer HTML-Datei besonders gut als Baum repräsentieren.

In der folgenden Grafik sehen sie auf der linken Seite HTML und auf der rechten Seite die als Baum dargestellte Hierarchie des Dokuments.

```
<div>
    <h1>Überschrift</h1>
    <p>Text</p>
    <div>
        <p>Mehr Text</p>
        <strong>Fetter Text</strong>
    </div>
    <i> Kursiver Text </i>
</div>
```



(a) Tags; 1 Punkt
Erstellen sie eine *Enumeration* `Tag`, der die gängigen HTML Tags `P`, `STRONG`, `II`[2], `DIV` und `H1` enthält.

(b) Baumstruktur; 2 Punkte
Nun wollen wir die Struktur des HTML-Dokuments als Baum modellieren und nehmen dafür zwei Vereinfachungen vor:

- Die Tags `<p>`, `<strong>`, `<i>` und `<h1>` können nur Text, aber keine weiteren Tags enthalten
- `<div>` Tags können (beliebig viele) andere Tags enthalten, jedoch keinen direkten Text

Erstellen sie eine Datenklasse `Node` mit zwei Attributen. Das erste Attribut `tag` soll den Tag des Knotens bestimmen den zuvor definierten Aufzählungstyp `Tag` verwenden. Das zweite Attribut `children` kann entweder eine Zeichenkette oder eine Liste sein. Als Zeichenkette stellt es den Text innerhalb von den Textknoten `<p>`, `<strong>`, `<i>` oder `<h1>`) dar, als Liste von `Node`s bestimmt es die Kinder eines Knotens. Das besondere an dieser Darstellung ist, dass durch die Liste nicht nur zwei, sondern beliebig viele Kinder ermöglicht werden.

Das Beispiel von oben kann wie folgt in die Baumstruktur übertragen werden:

```
>>> tree = Node(Tag.DIV, [
...     Node(Tag.H1, "Überschrift"),
...     Node(Tag.P, "Text"),
...     Node(Tag.DIV, [
...         Node(Tag.P, "Mehr Text"),
...         Node(Tag.STRONG, "Fetter Text")
...     ]),
...     Node(Tag.II, "Kursiver Text")
... ])
```

(c) Validierung; 3.5 Punkte
Die Baumdarstellung unserer HTML-Dateien erlaubt es Text in `<div>`-Tags zu

[2] Um mit Flake8 kompatibel zu sein verwenden wir für den Tag `<i>` das Enumerations-Feld `II` statt `I`.

schreiben oder den Textelementen `<p>`, `<strong>`, `<i>` oder `<h1>` weitere Tags als Kinder hinzuzufügen.

Schreiben Sie eine Funktion `validate`, die einen HTML-Baum `tree` als Argument nimmt und einen Wahrheitswert zurückgibt, der angibt ob die Vereinfachungen aus Aufgabe (b) eingehalten werden. **Verwenden Sie Rekursion und Pattern Matching!**

Zum Beispiel:

```
>>> validate(Node(Tag.II, "Hallo Welt"))
True
>>> validate(Node(Tag.DIV, "Hallo Welt"))
False
>>> validate(Node(Tag.DIV, [Node(Tag.H1, "Top-Level")]))
True
>>> validate(Node(Tag.DIV, [Node(Tag.H1, "Ganz Oben"),
↪  Node(Tag.STRONG, ""), Node(Tag.DIV, [])]))
True
>>> validate(Node(Tag.DIV, [Node(Tag.H1, "Ganz Oben"),
↪  Node(Tag.STRONG, [Node(Tag.P, "Hallo")])]))
False
>>> validate(Node(Tag.DIV, [Node(Tag.H1, "Ganz Oben"),
↪  Node(Tag.STRONG, ""), Node(Tag.DIV, "Schwimm Dori!")]))
False
```

(d) Konvertierung; 3.5 Punkte
Schreiben Sie eine Funktion `tree_to_html`, die einen HTML-Baum `tree` und die aktuelle Einrückugstiefe (Anzahl der Leerzeichen vor dem ersten Symbol der Zeile) `indent` als Argument nimmt und einen `str` zurückgibt, der das entsprechende HTML-Dokument darstellt. Die Einrückungstiefe soll in den Kindern eines `<div>` Knotens um 2 erhöht werden. **Verwenden Sie Rekursion und Pattern Matching!**

```
>>> tree_to_html(Node(Tag.II, "Hallo Welt"), 0)
'<i>Hallo Welt</i>'
>>> tree_to_html(Node(Tag.H1, "Hallo Welt"), 4)
'␣␣␣␣<h1>Hallo Welt</h1>'
>>> tree_to_html(Node(Tag.DIV, [Node(Tag.P, "Top-Level")]), 0)
'<div>\n␣␣<p>Top-Level</p>\n</div>'
>>> tree_to_html(tree, 1)
'␣<div>\n␣␣␣<h1>Überschrift</h1>\n␣␣␣<p>Text</p>\n␣␣␣<div>\n␣␣␣␣␣<p>Mehr
↪  Text</p>\n␣␣␣␣␣<strong>Fetter
↪  Text</strong>\n␣␣␣</div>\n␣␣␣<i>Kursiver Text</i>\n␣</div>'
```

Hinweis: In den oberen Beispielen haben wir zur besseren Visualisierung *alle* Leerzeichen graphisch hervorgehoben. Verwenden Sie in Ihrer Funktion das normale Leerzeichen `" "`

(e) HTML-Datei erstellen und anzeigen (optional)
Abschließend können wir den resultierenden HTML-Quelltext in eine Datei schreiben und im Browser anzeigen. Hierzu können Sie die Funktion `write_html` verwenden, die wir bereits für Sie definiert haben. Sie nimmt einen beliebigen HTML-Baum `html_tree` und einen Dateipfad `filepath` (default-Wert: `index.html`) als Argumente und schreibt den HTML-Quelltext in den angegebene Dateipfad.

```python
def write_html(html_tree: Node, filepath: str = "index.html"):
    if not validate(html_tree):
        print("Der HTML Baum ist ungültig!")
        return
    with open(filepath, 'w+') as file:
        file.write(tree_to_html(html_tree, 0))
    print(f"Die HTML Datei wurde unter {filepath} gespeichert.")
```

Sie können die Datei in einem Browser Ihrer Wahl anzeigen lassen, in dem Sie die Datei mit einem Doppelklick über den Dateimanager Ihres Computers öffnen. Das Beispiel von oben wird wie folgt angezeigt:

Überschrift

Text

Mehr Text

Fetter Text
Kursiver Text

Hier können Sie Ihrer Kreativität freien Lauf lassen :) Probieren Sie gerne die verschiedenen Tags aus und erstellen Sie Ihre eigene HTML-Datei.

Aufgabe 7.4 (Erfahrungen; 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 für 7 Stunden 30 Minuten.

Der Build-Server überprüft ebenfalls, ob Sie das Format korrekt angegeben haben. Prüfen Sie, ob der Build-Server mit Ihrer Abgabe zufrieden ist, so wie es im Video zur Lehrplattform gezeigt wurde.