Tutorium 08

Exercise 07 - Korrektur

Objekt-orientierte Programmierung

Exercise 08 - Hilfe und Fragen

Exercise 07 - Musterlösung

Aufgabe 7.1 a)

```python
def ask(s: str) -> Optional[bool]:
    match input(f"{s}? [Yes / No]: "):
        case "Yes" | "yes":
            return True
        case "No" | "no":
            return False
        case _:
            return None
```

- Default-Case `case _:` oder exhaustive pattern matching
- Keinen unnötigen Code-Duplikat mit `|`

Aufgabe 7.1 b)

```python
class Operator(Enum):
    ADD = auto()
    MUL = auto()


def eval[T: (int, str)](t: tuple[Operator, T, T]) -> Optional[T]:
    match t:
        case (Operator.ADD, x, y):
            return x + y
        case (Operator.MUL, int(i), int(j)):
            return i * j
        case _:
            return None
```

- Default-Case `case _:` oder exhaustive pattern matching
- Generic `T` mit der Einschränkung `T: (int, str)` (nicht `int | str`!)

Aufgabe 7.1 c)

```python
@dataclass
class Cons[T]:
    head: T
    tail: Optional["Cons[T]"] = None


type LList[T] = Optional[Cons[T]]
```

Hierbei gab es einige Verwirrung, was wahrscheinlich unter anderem an der Benennung lag

Aufgabe 7.1 c)

```python
@dataclass
class Node[T]:
    value: T
    next: Optional['Node[T]'] = None


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

Wobei jede `Node` auch einen Wert vom Typ `T` hat und weiß was die nächste `Node` in der Liste ist. Also ist `Node` quasi ein Eintrag in der Liste, der sich merkt was als nächstes kommt und einem Wert zugeordnet wird.

Aufgabe 7.1 c)

```
def tail[T](xs: LList[T]) -> LList[T]:
    match xs:
        case None:
            return None
        case Cons(_, tail):
            return tail

def len(xs: LList[Any]) -> int:
    match xs:
        case None:
            return 0
        case Cons(_, tail):
            return 1 + len(tail)
```

das geht aber auch schöner:

```
def next[T](xs: LinkedList[T]) -> LinkedList[T]:
    return xs.next if xs else xs

def len(xs: LinkedList[Any]) -> int:
    return 1 + len(xs.next) if xs else 0
```

Aufgabe 7.2 - BTree Definition

```python
@dataclass
class Node[T]:
    mark: T
    left: Optional["Node[T]"] = None
    right: Optional["Node[T]"] = None


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

- Binärbaum
- generisch und rekursiv definiert wie unsere `LList[T]`

Aufgabe 7.2 a)

```
def contains[T](tree: BTree[T], val: T) -> bool:
    return tree and (tree.m == val or contains(tree.left, val) or contains(tree.right, val))
```

- wir testen ob der aktuelle `tree` eben `None` ist
- dann testen wir ob der aktuelle Wert unserem gesuchten entspricht
- Wenn das nicht der Fall ist laufen wir den Baum rekursiv ab
- Wichtig ist dass man Links **und** Rechts abläuft, und nicht einen Zweig vergisst

Aufgabe 7.2 b)

```python
def leaves[T](tree: BTree[T]) -> list[T]:
    match tree:
        case None:
            return []
        case Node(mark, None, None):
            return [mark]
        case Node(_, left, right):
            return leaves(left) + leaves(right)
```

- exhaustive matchen oder default-case!

Aufgabe 7.2 - AST Definition

```
type AST = BTree[int | str]
```

- Schlechtes Design, weil `str` kann alles sein, wir gehen nur davon aus dass es `"+"` oder `"*"` ist.
- Man hätte auch einfach den `Enum` aus Aufgabe 1 verwenden können...

```
class Op(Enum):
    ADD = "+"
    MUL = "*"

type AST = BTree[int | Op]
```

Aufgabe 7.2 c)

```python
def evaluate(tree: AST) -> Optional[int]:
    match tree:
        case Node(int(i), None, None):
            return i
        case Node("*" | "+", left, right):
            left = evaluate(left)
            right = evaluate(right)
            if left is None or right is None:
                return None
            if tree.mark == "+":
                return left + right
            else:
                return left * right
        case _:
            return None
```

```python
def evaluate(tree: AST) -> Optional[int]:
    match tree:
        case Node(int(i), None, None):
            return i
        case Node(Op.MUL | Op.ADD, left, right) if left and right:
            left = evaluate(left)
            right = evaluate(right)
            if tree.mark == Op.ADD:
                return left + right
            else:
                return left * right
        case _:
            return None
```

Aufgabe 7.2 d)

```
def infix_str(tree: AST) -> str:
    match tree:
        case Node(int(i), _, _):
            return str(i)
        case Node(str(s), left, right):
            return f"({infix_str(left)} {s} {infix_str(right)})"
        case _:
            return ""
```

- Das wichtigste war die Reihenfolge der Rekursiven aufrufe.
- Hier also `infix_str(left) + s + infix_str(right)`

```python
def prefix_str(tree: AST) -> str:
    match tree:
        case Node(int(i), _, _):
            return str(i)
        case Node(str(s), left, right):
            return f"({s} {prefix_str(left)} {prefix_str(right)})"
        case _:
            return ""

def postfix_str(tree: AST) -> str:
    match tree:
        case Node(int(i), _, _):
            return str(i)
        case Node(str(s), left, right):
            return f"({postfix_str(left)} {postfix_str(right)} {s})"
        case _:
            return ""
```

Fragen?

Probleme bei Exercise 07

- Generics `T` wurden unnötig verwendet
  - Ihr müsst keine Generics verwenden wenn nicht nötig
  - Sowas wie `T: int` ist unnötig, schreibt einfach direkt `int`
- Rekursion als Konzept - dabei hilft nur Üben
  - begegnet einem aber auch unglaublich selten in der echten Welt
  - wenn es Verständnis-Fragen gibt einfach melden
- `@dataclass` mit `Enum` verwendet (dabei geht alles kaputt)

Objekt orientiertes Programmieren - OOP

Definitionen in der OOP - Klassen

- Eine Klasse ist wie ein Bauplan
- Jede Klasse definiert die Eigenschaften und das Verhalten
- Verhalten sind Methoden also `def`
- Eigenschaften sind Attribute, `int`, `float`, `str`, `list`, ...
- Die Eigenschaften definieren den Zustand
- Eigenschaften können sich ändern

Beispiel - Cat

```python
from dataclasses import dataclass

@dataclass
class Cat:
    age: int
    weight: float
    name: str

    def meow(self):
        print("Meow")
```

Beispiel - Dog

```python
from dataclasses import dataclass

@dataclass
class Dog:
    age: int
    weight: float
    name: str

    def woof(self):
        print("Woof")
```

Objekte erzeugen

```
dog = Dog(age=3, weight=50.0, name="dog")
cat = Cat(age=7, weight=4.5, name="cat")
dog.woof() # Woof
cat.meow() # Meow
```

Objekte erzeugen

```
dog = Dog(3, 50.0, "dog")
cat = Cat(7, 4.5, "cat")
dog.woof() # Woof
cat.meow() # Meow
```

Zustand ändern

```
cat.age = 8
print(cat.age) # 8
```

Vererbung

- Was haben `Cat` und `Dog` gemeinsam?
  - `age`
  - `weight`
  - `name`

=> Lösung ist eine Oberklasse `Animal`

Vererbung - Animal

```python
@dataclass
class Animal:
    age: int
    weight: int
    name: str
```

```python
@dataclass
class Cat(Animal):
    pass

@dataclass
class Dog(Animal):
    pass
```

Bisher in der Vorlesung

```
def make_noise(animal: Animal):
    match animal:
        case Dog():
            print("Woof")
        case Cat():
            print("Meow")
        case _:
            pass
```

Bisher in der Vorlesung

```
def make_noise(animal: Animal):
    match animal:
        case Dog():
            print("Woof")
        case Cat():
            print("Meow")
        case _:
            pass
```

das ist kein gutes Design, warum?

was ist wenn wir jetzt eine neue Klasse `Mouse` erstellen wollen.

```
@dataclass
class Mouse(Animal):
    pass
```

jetzt müssen wir `Mouse` zu `make_noise` hinzufügen

```
def make_noise(animal: Animal):
    match animal:
        case Dog():
            print("Woof")
        case Cat():
            print("Meow")
        case Mouse():
            print("Peep")
        case _:
            pass
```

das kann ziemlich nervig werden. Vor allem wenn wir größere Projekte haben

Polymorphism

> Die Polymorphie der objektorientierten Programmierung ist eine Eigenschaft, die immer im Zusammenhang mit Vererbung und Schnittstellen (Interfaces) auftritt. Eine Methode ist polymorph, wenn sie in verschiedenen Klassen die gleiche Signatur hat, jedoch erneut implementiert ist.

```python
@dataclass
class Animal:
    age: int
    weight: float
    name: str

    def make_noise(self) -> None:
        pass

@dataclass
class Cat(Animal):

    def make_noise(self) -> None:
        print("Meow")

@dataclass
class Dog(Animal):

    def make_noise(self) -> None:
        print("Woof")
```

Geht das schöner in Python 3.12?

Ja, mit `override` sagt sogar pylance bescheid wenn wir eine polymorphe Methode falsch überschrieben haben!

```
form dataclasses import dataclass
from typing import override

@dataclass
class Cat(Animal):

    @override
    def make_noise(self) -> None:
        print("Meow")
```

Unsere neue polymorphe Methode benutzen

```python
@dataclass
class Zoo:
    animals: list[Animal] = []


zoo = Zoo(animals=[
            Cat(age=6, weight=4.5, name="Milow"),
            Cat(age=7, weight=5.0, name="Blacky"),
            Dog(age=12, weight=40.3, name="Bernd")
        ])

for animal in zoo.animals:
    animal.make_noise() # Meow
                        # Meow
                        # Woof
```

Oberklasse Animal

- Was passiert wenn man `Animal` euzeugt?
  - Wir haben ein Objekt was eigentlich nicht erzeugt werden sollte
  - Es ist eine Art Schablone für alle Klassen von Typ `Animal`

Können wir die Instanziierung verhindern? Ja!

Abstrakte Klassen

- abstrakte Klassen sind kein konkreter Bauplan für ein Objekt
- stattdessen eine Schablone für weiter Baupläne
- `Animal` ist so einen Schablone für alle Tiere

Abstrakte Klassen - in anderen Programmiersprachen

Beispiel Java mit `abstract`

```
abstract class Animal {
    private int age;
    private float weight;
    private String name;

    abstract void make_noise();
}

class Cat extends Animal {
    @override
    void make_noise() {
        System.out.println("Meow")
    }
}
```

Abstrakte Klassen - Python

- `ABC` aus dem Modul `abc`.
- Ja sie mussten `abstract` wirklich abkürzen mit `ABC` (Abstract Base Class)
- für abstrakte Methoden gibt es die Annotation `@abstractmethod` im selben Modul

```python
from abc import ABC, abstractmethod

class Animal(ABC):
    age: int
    weight: float
    name: str

    @abstractmethod
    def make_noise():
        pass
```

Fragen?

Exercise 08