Einführung in die Programmierung

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

Universität Freiburg
Institut für Informatik
Wintersemester 2023

Übungsblatt 7

Abgabe: Montag, 04.12.2023, 9:00 Uhr morgens

Aufgabe 7.1 (Pattern Matching und Enums; 10 Punkte; Datei: `match.py`)

In dieser Aufgabe sollen Sie Pattern-Matching benutzen. Fassen Sie mehrere `case`-Anweisungen zusammen, falls diese den gleichen Fall abdecken. Sie dürfen desweiteren **keine** if-Statements benutzen. Verwenden Sie Typvariablen, falls dies für die Typannotation der jeweiligen Teilaufgabe notwendig ist.

(a) Matchen auf Literalen (3 Punkte)
Schreiben Sie eine Funktion `ask`, die einen String `question` als Argument nimmt. Das Programm soll die Benutzerin nach einer Antwort fragen. Ist die Antwort `Yes` oder `yes`, soll `True` *zurückgegeben* werden. Ist die Antwort `No` oder `no`, soll `False` *zurückgegeben* werden. Bei allen anderen Antworten soll `None` *zurückgegeben* werden. In Ihrem Programm darf bis auf `question` keine Variable vorkommen. Achten Sie darauf, dass dich Ihr Programm genau wie folgt verhält:

```
>>> ask("Are you here")
Are you here? [Yes / No]: Yes
True
>>> ask("Are you here")
Are you here? [Yes / No]: no
False
>>> ask("Are you here")
Are you here? [Yes / No]: kjhsdfgs
```

(b) Operator Enum (1 Punkt)
Definieren Sie ein `Enum Operator`, das entweder `ADD` oder `MUL` sein kann.

(c) Auswerten eines arithmetischen Ausdrucks (2 Punkte)
Schreiben Sie eine Funktion `eval`, die ein 3-Tupel `t` als Argument nimmt. Das erste Element des Tupels soll ein Operator sein. Das zweite und dritte Element können ein String oder ein Integer sein, müssen aber stets den selben Typ besitzen. Das erste Element des Tupels bestimmt, welcher Operator auf die beiden anderen Elemente angewendet werden soll. Ist das erste Element der Operator `ADD` und die zwei letzten Elemente Zahlen, sollen die zwei Zahlen addiert und das Ergebnis zurückgegeben werden. Sind die beiden letzten Elemente Strings, sollen diese verkettet und das Ergebnis zurückgegeben werden. Ist das erste Element der Operator `MUL` und die anderen beiden Zahlen, soll deren Produkt zurückgegeben werden. Trifft keiner der Fälle ein, soll `None`

zurückgegeben werden.

```
assert eval((Operator.ADD, 2, 5)) == 7
assert eval((Operator.MUL, 2, 5)) == 10
assert eval((Operator.ADD, "2", "5")) == "25"
assert eval((Operator.MUL, "2", "5")) is None
```

(d) LLists (1 Punkt)
In der Vorlesung haben Sie die in Python vordefinierten Listen kennengelernt. In dieser Aufgabe schreiben Sie eine Variation von diesem Datentyp der ähnlich wie Bäume definiert ist.
Definieren Sie dazu eine generische Datenklasse `Cons[T]`, die zwei Attribute hat. Das erste Attribut `head` ist ein Element von Typ `T`. Das zweite Attribut `tail` ist wieder ein `Cons[T]`[1]. Das zweite Attribut soll per default[2] `None` sein. Definieren Sie dann einen generischen Typealias `LList[T]`[3] der für `None` oder `Cons[T]` steht.

```
example = Cons(1, Cons(2, None))
assert example.head == 1
assert example.tail == Cons(2, None)
assert example.tail
assert example.tail.head == (2)
assert example.tail.tail is None
```

(e) tail für LLists (1.5 Punkte)
Schreiben Sie eine Funktion `tail`, die eine LList `xs` als Argument nimmt. Die Funktion soll eine LList zurückgeben, die alle Elemente bis auf das Erste von `xs` beinhaltet.

```
assert tail(None) is None
assert tail(Cons(1, None)) is None
assert (tail(Cons("1", Cons("2", None)))
    == Cons("2", None))
assert (tail(Cons("1", Cons("2", Cons("5", None))))
    == Cons("2", Cons("5", None)))
```

(f) len für LLists(1.5 Punkte)
Schreiben Sie eine Funktion `len`, die eine LList `xs` von beliebigen Elementen als Argument nimmt und die Länge dieser LList zurückgibt.[4]

```
assert len(None) == 0
assert len(Cons(True, None)) == 1
assert len(Cons(True, Cons(False, None))) == 2
assert len(Cons(True, Cons(False, Cons(True, None)))) == 3
```

[1] Wenn man in Python in einer Klassendefinition die Klasse als Typ angeben will, muss man den Typ in Anführungszeichen schreiben. Vergleichen Sie dies mit der Implementation von BTree.

[2] Sie können Standardwerte an Attribute zuweisen, indem Sie `= <Wert>` hinter das Attribut schreiben. Wenn Sie dann eine Instanz dieser Klasse erstellen, müssen Sie für default Attribute keine Werte übergeben.

[3] Kurz für Linked List

[4] Beachten Sie, dass dies nicht genau der 'Höhe' des Baumes entspricht, wie sie in der Vorlesung definiert wurde.

Aufgabe 7.2 (Binäre Bäume; Punkte: 10; Datei: `trees.py`)

In dieser Aufgabe sollen Sie verschiedene Funktionen für Bäume schreiben. Verwenden Sie dafür die in der Vorlesung vorgestellte Definition für generische (Binär)bäume (`BTree[T]`).

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

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

Verwenden Sie Typvariablen, falls dies für die jeweilige Typannotation notwendig ist.

(a) `contains`; 2 Punkte
Schreiben Sie eine Funktion `contains`, die einen beliebigen Baum `tree` und einen Wert `val` als Argumente nimmt und zurückgibt, ob `val` als Markierung in `tree` vorkommt oder nicht. Nehmen Sie an, dass der Typ der Baummarkierungen mit dem von `val` übereinstimmt.

```
tree = Node(5, Node(4), Node(1, Node(0), Node(2)))
assert contains(tree, 5)
assert contains(tree, 2)
assert not contains(tree, 3)
assert not contains(None, 42)
```

(b) `leaves`; 2 Punkte
Schreiben Sie eine Funktion `leaves`, die einen beliebigen Baum `tree` als Argument nimmt und eine Liste der Markierung aller Blätter des Baumes (von links nach rechts) zurückgibt.

```
assert leaves(Node(5, Node(4), Node(1, Node(0), Node(2)))) == [4, 0, 2]
assert leaves(Node(1, Node(2, Node(3, Node(4, Node(5)))))) == [5]
assert leaves(None) == []
```

(c) `evaluate`; 3 Punkte
Schreiben Sie eine Funktion `evaluate`, die einen Ausdrucksbaum `tree` wie in dem Beispiel aus der Vorlesung[5] als Argument nimmt, und entweder dessen numerischen Wert berechnet und zurückgibt, oder `None` zurückgibt, falls `tree` keinen validen Ausdrucksbaum darstellt. Beachten Sie, dass der Baum sowohl Markierungen vom Typ `int` als auch Markierungen vom Typ `str` besitzen kann. Gültige Operatoren sind (nur) `"+"` und `"*"`, welche wie üblich als Addition bzw. Multiplikation definiert sind.

```
example = Node("+", Node(4),
                    Node("*", Node("+", Node(1), Node(2)), Node(2)))
assert evaluate(example) == 10
```

[5] https://proglang.informatik.uni-freiburg.de/teaching/info1/2023/lecture/infoI10.pdf, Folie 16

```
example2 = Node("+", Node(4),
                     Node("*", Node("+", Node(1), None), Node(2)))
                     assert evaluate(example2) is None
example3 = Node("+", Node(4),
                     Node("*", Node("-", Node(2), Node(1)), Node(2)))
assert evaluate(example3) is None
```

(d) Traversierungen; 3 Punkte
Schreiben Sie die 3 Funktionen `prefix_str`, `infix_str` und `postfix_str`, die jeweils einen Ausdrucksbaum wie aus Teil c) als Argument nehmen und den Ausdruck als String in Präfix-, Infix-, bzw. Postfixnotation zurückgeben. Diese erhält man durch Pre-, In-, bzw. Post-Order Traversierung über den Baum. Klammern Sie jede Operation explizit und separieren Sie Werte und Operatoren jeweils mit einem Leerzeichen. Sie dürfen davon ausgehen, dass der Ausdrucksbaum einen validen Ausdruck beschreibt.

```
example = Node("+", Node(4),
                    Node("*", Node("+", Node(1), Node(2)), Node(2)))
assert prefix_str(example) == "(+ 4 (* (+ 1 2) 2))"
assert infix_str(example) == "(4 + ((1 + 2) * 2))"
assert postfix_str(example) == "(4 ((1 2 +) 2 *) +)"
assert (prefix_str(Node(0)) == infix_str(Node(0)) == postfix_str(Node(0))
    == "0")
assert prefix_str(None) == infix_str(None) == postfix_str(None) == ""
```

Aufgabe 7.3 (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 `3.5 h` steht dabei für 3 Stunden 30 Minuten.