Informatik I: Einführung in die Programmierung

Prof. Dr. Peter Thiemann
Hannes Saffrich
Wintersemester 2020

Universität Freiburg
Institut für Informatik

Übungsblatt 7

Abgabe: Montag, 21.12.2020, 9:00 Uhr morgens, über git[1]

Hinweis

Aufgabenteile werden mit **0 Punkten** bewertet wenn:

- Dateien und Definitionen nicht so benannt sind, wie im Aufgabentext gefordert;
- Dateien falsche Formate haben, z.B. PDF statt plaintext; oder
- Pythonskripte wegen eines Syntaxfehlers nicht ausführbar sind.

Es gibt **Punktabzug** wenn:

- Funktionen keine oder falsche Typannotationen aufweisen. Ausnahme: Bei Funktionen, die stets `None` zurückgeben, kann der Rückgabetyp weggelassen werden.

Gruppenaufgaben müssen von allen Mitgliedern abgegeben werden und in der ersten Zeile müssen die Mitglieder in einem Kommentar vermerkt werden, z.B:

```
# Gruppe: xy123, yz56, zx934
```

Hinweis

Hier ist eine Übersicht der bisherigen Typannotationen:

```
from typing import Union, Optional
x: int              = 42
x: float            = 42.0
x: complex          = 42.0 + 23.0j
x: bool             = True
x: str              = 'foo'
x: type(None)       = None
x: list[int]        = [1, 2, 3]
x: list[list[int]] = [[1, 2, 3], [4, 5]]
x: MyClass          = MyClass(...)  # Angenommen MyClass wurde definiert
x: Union[int,bool] = 42
x: Union[int,bool] = True
x: Optional[int]   = None          # Kurzform für Union[int, type(None)]
```

Für komplexere `Union`s empfiehlt es sich zur Lesbarkeit Typ-Aliase zu definieren:

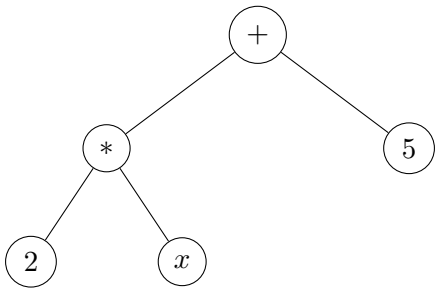
```
MyType = Union[int, bool, list[int], MyClass, type(None)]
def combine(x: MyType, y: MyType) -> MyType:
    # ...
```

[1] `https://inpro.informatik.uni-freiburg.de/`

Aufgabe 7.1 (Symbolische Arithmetik; Datei: `optimizer.py`; Punkte: 18)

In dieser Aufgabe werden wir uns mit Ausdrucksbäumen einer kleinen, arithmetischen Sprache beschäftigen. Die Sprache soll dabei Variablen, ganze Zahlen, Addition und Multiplikation unterstützen.

Der Baum für den Ausdruck $2 * x + 5$ sieht z.B. wie folgt aus:



Mit der `Node`-Klasse aus der Vorlesung[2], können wir den Baum als folgende Instanz darstellen:

```
example = Node('+', Node('*', leaf(2),
                              leaf('x')),
                    leaf(5))
```

In dieser Aufgabe sollen Sie einen Algorithmus implementieren, der Ausdrücke dieser arithmetischen Sprache optimiert. Zum Beispiel soll für die Eingabe `(x + x) + (x + x)` folgende Ausgabe erzeugt werden:

```
> (x + x) + (x + x)
= (2 * (x + x))
= (2 * (2 * x))
= ((2 * 2) * x)
= (4 * x)
```

Solche Transformationen finden auch in echten Compilern statt und laufen grob in 3 Schritten ab:

- Die Eingabe ist ein String, der durch einen sogenannten *Parser* in einen Ausdrucksbaum umgewandelt wird, falls möglich.
- Ist die Umwandlung erfolgreich, so wird dieser Baum in mehreren Schritten transformiert.
- Wenn keine weiteren Transformationen mehr möglich sind, wird der transformierte Ausdrucksbaum wieder zu einem String umgewandelt und ausgegeben.

Für das Einlesen eines Ausdrucksbaumes im ersten Schritt, stellen wir Ihnen die

[2]Die `Node`-Klasse aus der Vorlesung und alle dafür definierten Funktionen finden Sie in der Datei `tree.py` unter `http://proglang.informatik.uni-freiburg.de/teaching/info1/2020/exercise/sheet07/tree.py`

Funktion `parse` aus der Datei `expr_parser.py` zu Verfügung[3]. Sie müssen nicht verstehen wie diese Funktion implementiert ist, sondern nur wie sie verwendet werden kann. Beispiele:

```
from tree import Node, leaf
from expr_parser import parse

#   def parse(source_code: str) -> Optional[Node]: [...]

assert parse("2 * x + 5") == example
assert parse("invalid input") == None
```

Die restlichen beiden Schritte müssen Sie in den folgenden Aufgabenteilen implementieren. Jeder Aufgabenteil gibt 2 Punkte; (a) bis (f) sind unabhängig voneinander; (g) bis (i) bauen aufeinander und auf den vorherigen Aufgabenteilen auf.

(a) Schreiben Sie Funktionen `is_var`, `is_val` und `is_op`, die jeweils einen Ausdrucksbaum als Argument nehmen und zurückgeben, ob es sich um eine Variable, eine Zahl oder einen Baum handelt, der einen Operator als Wurzelknoten hat. Schreiben Sie die Funktionen `is_add` und `is_mul`, die sich wie `is_op` verhalten, aber nur dann `True` zurückgeben, wenn es sich bei dem Operator um eine Addition bzw. eine Multiplikation handelt. Beispiele:

```
assert     is_var(leaf('x'))
assert not is_var(leaf(5))
assert     is_val(leaf(5))
assert not is_val(leaf('x'))
assert     is_op(Node('*', leaf(5), leaf(6))
assert not is_op(leaf('x'))
assert     is_op(example)
assert     is_add(example)
assert not is_mul(example)
```

In den darauffolgenden Aufgabenteilen helfen diese Funktionen um zu unterscheiden was für einen Baum man gerade vor sich hat.

(b) Schreiben Sie eine Funktion `show_node`, die einen Ausdrucksbaum als Argument nimmt und den zugehörigen String in Infix-Notation zurückgibt. Machen Sie die Klammerung von Operatoren explizit und verwenden Sie genau ein Leerzeichen um Operatoren von Argumenten zu trennen. Beispiel:

```
assert show_node(example)   == '((2 * x) + 5)'
assert show_node(leaf('x')) == 'x'
assert show_node(leaf(2))   == '2'
```

(c) Schreiben Sie eine Funktion `show_node_prefix`, die einen Ausdrucksbaum als Argument nimmt und den zugehörigen String in Prefix-Notation zurückgibt.

[3]Den Code gibt es unter `http://proglang.informatik.uni-freiburg.de/teaching/info1/2020/exercise/sheet07/expr_parser.py`

Die Prefix-Notation ist auch ohne Klammern eindeutig. Beispiel:

```
assert show_node_prefix(example) == '+ * 2 x 5'
assert show_node_prefix(parse('1 + 2')) == '+ 1 2'
assert show_node_prefix(parse('(1 * 2) + (3 * 4)')) == '+ * 1 2 * 3 4'
```

(d) Schreiben Sie eine Funktion `opt_times_two`, die einen Ausdrucksbaum `e` als Argument nimmt, und wenn `e` eine Addition ist, deren linker und rechter Teilbaum gleich[4] sind, den Baum durch eine Multiplikation mit `2` ersetzt. Für andere Bäume soll `None` zurückgegeben werden. Beispiele:

```
assert opt_times_two(parse('x + x')) == parse('2 * x')
assert opt_times_two(parse('(5 * 3) + (5 * 3)')) == parse('2 * (5 * 3)')
assert opt_times_two(parse('x + y')) == None
assert opt_times_two(parse('x + (x + x)')) == None
```

Wir führen für diese Transformation folgende Notation ein:

$$e + e \implies 2 * e$$

Die Variable e steht dabei für einen beliebigen Teilbaum.

(e) Schreiben Sie eine Funktion `opt_assoc`, die den Transformationsregeln

$$e_1 \odot (e_2 \odot e_3) \implies (e_1 \odot e_2) \odot e_3$$

entspricht, also dem Assoziativgesetz. Der Operator $\odot$ steht dabei für einen beliebigen Operator. Beispiele:

```
assert opt_assoc(parse('x * (y * z)')) == parse('(x * y) * z')
assert opt_assoc(parse('x + (y + z)')) == parse('(x + y) + z')
assert opt_assoc(parse('x + (y * z)')) == None
assert opt_assoc(parse('(x + y) + z')) == None
```

(f) Schreiben Sie eine Funktion `opt_eval`, die einen Ausdrucksbaum `e` als Argument nimmt, und wenn `e` eine Operation auf zwei Konstanten ist, die Rechenoperationen ausführt. Ansonsten soll `None` zurückgegeben werden. Beispiele:

```
assert opt_eval(parse('2 * 3')) == parse('6')
assert opt_eval(parse('2 + 3')) == parse('5')
assert opt_eval(parse('2 * x')) == None
assert opt_eval(parse('2 * (3 * 4)')) == None
```

Schreiben und verwenden Sie hierzu eine Hilfsfunktion

```
apply_op(op: str, l: int, r: int) -> Optional[int]
```

die Falls `op` die Stringrepräsentation eines Operators ist, diesen auf `l` und `r` anwendet und das Ergebnis zurückgibt. Ansonsten soll `None` zurückgegeben werden. Beispiele:

[4] Da `Node` als `dataclass` definiert ist, macht der `==` Operator genau das Richtige: es reicht aus wenn es die gleichen Bäume sind; es müssen nicht die selben Bäume sein.

```
assert apply_op('+', 2, 3) == 5
assert apply_op('*', 2, 3) == 6
assert apply_op('x', 2, 3) == None
```

(g) Schreiben Sie eine Funktion `opt_any`, die einen Ausdrucksbaum `e` als Argument nimmt, der Reihe nach versucht die Optimierungen `opt_times_two`, `opt_assoc` und `opt_eval` anzuwenden. Glückt eine der Optimierungen, so soll der optimierte Ausdrucksbaum zurückgegeben werden, ansonsten soll versucht werden die nächste Optimierung anzuwenden.

Wenn alle Optimierungen fehlgeschlagen sind, dann soll überprüft werden ob es sich um einen Operatorknoten handelt. Ist dies der Fall, so soll versucht werden `opt_any` auf die Teilbäume anzuwenden: erst auf den linken Teilbaum, wenn dies fehlschlägt, auf den rechten Teilbaum, wenn dies auch fehlschlägt, dann soll `None` zurückgegeben werden. Ist die Optimierung eines Teilbaumes erfolgreich, so soll der Teilbaum durch seine optimierte Version ersetzt werden und der so entstehende Baum zurückgegeben werden.

Dies erlaubt es uns die einzelnen Optimierungsschritte anzuzeigen, wie im Beispiel der Einleitung.

Beispiele:

```
assert opt_any(parse('(x + x) + (x + x)')) == parse('(2 * (x + x))')
assert opt_any(parse('2 * (x + x)')) == parse('(2 * (2 * x))')
assert opt_any(parse('(2 * (2 * x))')) == parse('((2 * 2) * x)')
assert opt_any(parse('((2 * 2) * x)')) == parse('(4 * x)')
assert opt_any(parse('(4 * x)')) == None
```

(h) Schreiben Sie eine Funktion `opt_all`, die einen Ausdrucksbaum `e` als Argument nimmt, diesen so lange mit `opt_any` transformiert bis keine Optimierung mehr greift und dann eine Liste aller Zwischenergebnisse inklusive `e` zurückgibt.

Beispiel:

```
assert opt_all(parse('(x + x) + (x + x)')) == [
         parse('(x + x) + (x + x)'),
         parse('(2 * (x + x))'),
         parse('(2 * (2 * x))'),
         parse('((2 * 2) * x)'),
         parse('(4 * x)')
       ]
```

(i) Schreiben Sie diese Teilaufgabe in eine neue Datei `optimizer_repl.py`. Importieren und verwenden Sie die Funktionen `parse`, `opt_all` und `show_node`, um eine "Optimizer REPL"[5] zu implementieren, wie im Einleitungsbeispiel beschrieben. Bevor die Benutzereingabe zu einem Baum umgewandelt wird, soll

[5]REPL steht für Read-Eval-Print-Loop und beschreibt Programme wie den interaktiven Pythoninterpreter oder die Kommandozeile, die in einer Endlosschleife eine Benutzereingabe einlesen (read), diese dann auswerten (eval) und das Ergebnis wieder ausgeben (print).

dabei überprüft werden, ob diese gleich `"quit"` ist und in diesem Fall das Programm beendet werden. Ungültige Benutzereingaben sollen ignoriert werden.

Beispiel:

```
> (x + x) + (x + x)
= ((x + x) + (x + x))
= (2 * (x + x))
= (2 * (2 * x))
= ((2 * 2) * x)
= (4 * x)

> 5 ( 23
Invalid input.

> (5 + 3) * (2 + 8)
= ((5 + 3) * (2 + 8))
= (8 * (2 + 8))
= (8 * 10)
= 80

> quit
Good bye!
```

Aufgabe 7.2 (Erfahrungen; Datei: `erfahrungen.txt`; 2 Punkte)

Notieren Sie hier Ihre Erfahrungen mit diesem Übungsblatt (benötigter Zeitaufwand, Probleme, Bezug zur Vorlesung, Interessantes, etc.).

Schreiben Sie den groben Zeitaufwand, den Sie für das *gesamte* Blatt benötigt haben, bitte wie folgt in die erste Zeile:

```
Zeitaufwand: 3.5h

[... Freitext, wie bisher ...]
```

Wenn sich genug Leute daran halten, dann können wir ein Pythonskript schreiben, welches automatisiert die durchschnittliche Bearbeitungszeit berechnet ;-)