Informatik I: Einführung in die Programmierung

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

Universität Freiburg
Institut für Informatik

Übungsblatt 7

Abgabe: Montag, 5.12.2022, 9:00 Uhr morgens

Exkursion: Pattern Guards

Für dieses Übungsblatt benötigen Sie eine Funktionalität des Pattern Matchings, die noch nicht in der Vorlesung behandelt wurde: die sogenannten *Pattern Guards*.

Ein Pattern Guard erweitert einen `case`-Zweig um eine boolsche Bedingung, die erfüllt sein muss, damit der `case`-Zweig ausgewählt wird. Beispiel:

```
some_point = (42, 23)
match some_point:
    case (42, y) if y > 0:  # The `if y > 0` is a pattern guard!
        print("Point has x = 42 and y is positive.")
    case (42, y):
        print("Point has x = 42 and y is negative or zero.")
    case (x, y):
        print("Point has x != 42.")
```

Pattern Guards sind sehr nützlich, da wenn die Bedingung falsch ist, versucht wird den nächsten `case`-Zweig auszuwählen. Würde man `y > 0` nicht als Pattern Guard abfragen, sondern innerhalb des Körpers des `case`-Zweiges, so würde etwas anderes passieren:

```
some_point = (42, 23)
match some_point:
    case (42, y):  # No pattern guard!
        if y > 0:
            print("Point has x = 42 and y is positive.")
    case (42, y):
        # This branch is unreachable, because if y <= 0,
        # then we've already entered the previous branch.
        print("Point has x = 42 and y is negative or zero.")
    case (x, y):
        print("Point has x != 42.")
```

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.

Ziel der Aufgabe ist es ein Programm zu schreiben, welches Ausdrücke dieser arithmetischen Sprache als Strings von der Kommandozeile einliest, diese in mehreren Schritten nach bestimmten Regeln umformt und dann wieder ausgibt. 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)
```

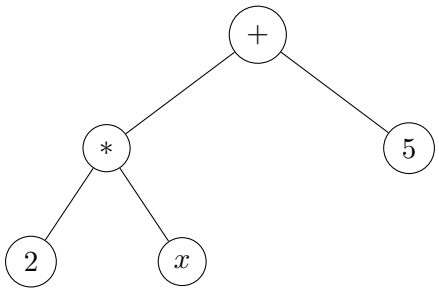
Solche Transformationen finden auch in echten Compilern statt, um den Code vor der Ausführung effizienter zu machen, und laufen grob in drei Schritten ab:

- Die Eingabe ist ein String, der durch einen sogenannten *Parser* in einen Ausdrucksbaum umgewandelt wird. Entspricht der Eingabestring keinem Ausdrucksbaum, so wird ein Syntax-Fehler ausgegeben.
- Der Ausdrucksbaum wird wiederholt umgeformt, wodurch ein neuer Ausdrucksbaum entsteht.
- Wenn keine weiteren Umformungen mehr möglich sind, wird der Ausdrucksbaum wieder zu einem String umgewandelt und ausgegeben.

Der Parser und die Datenstruktur für die Ausdrucksbäume sind bereits vorgegeben und auf unserer Webseite zu finden[1][2]. Die Umformungen der Ausdrucksbäume und das Konvertieren der Ausdrucksbäume zu Strings müssen Sie aber selbst implementieren.

Bevor wir zur eigentlichen Aufgabenstellung kommen, wollen wir aber zunächst noch etwas darüber nachdenken wie wir am besten die Ausdrucksbäume als Python-Datenstruktur modellieren.

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



Mit der `Node`-Klasse aus der Vorlesung, können wir diesen Baum wie folgt in Python modellieren:

```
e = Node(OpSym.ADD, Node(OpSym.MUL, leaf(2), leaf('x')), leaf(5))
```

Mit dieser Klasse wäre das Pattern Matching aber etwas unschön:

[1] http://proglang.informatik.uni-freiburg.de/teaching/info1/2022/exercise/sheet07/tree.py

[2] http://proglang.informatik.uni-freiburg.de/teaching/info1/2022/exercise/sheet07/parser.py

```
match e:
    case Node(int(i), None, None):
        # This case-branch matches leaf nodes of integer values.
    case Node(str(x), None, None):
        # This case-branch matches leaf nodes of variables.
    case Node(str(op), e1, e2):
        # This case-branch matches inner nodes of operators like
        # `e1 + e2` and `e1 * e2` for arbitrary sub-trees e1 and e2.
        # If we wouldn't match for variables before, it would also match variables...
```

Viel besser wäre es wenn wir einfach folgendes schreiben könnten:

```
e = Op(OpSym.ADD, Op(OpSym.MUL, Val(2), Var('x')), Val(5))
match e:
    case Val(i):
        # This case-branch matches leaf nodes of integer values.
    case Var(x):
        # This case-branch matches leaf nodes of variables.
    case Op(op, e1, e2):
        # This case-branch matches inner nodes of operators like
        # `e1 + e2` and `e1 * e2` for arbitrary sub-trees e1 and e2.
```

Um dies zu ermöglichen, definieren wir für jede Art von Knoten (Variable, `int`-Wert, Operator) eine eigene `dataclass` und fassen die Knoten dann als Union-Type zusammen. Da wir genau zwei Operatoren unterstützen (`+` und `*`) definieren wir für die Operator-Symbole einen Aufzählungstyp (`Enum`):

```
@dataclass
class Var:
    name: str

@dataclass
class Val:
    value: int

@dataclass
class OpSym(Enum):
    ADD = "+"
    MUL = "*"

@dataclass
class Op:
    sym: OpSym
    left: 'Node'
    right: 'Node'

Node = Var | Val | Op
```

Ein Objekt vom Typ `Node` ist also entweder

- ein Objekt vom Typ `Var`, das einen Variablennamen als String enthält;
- ein Objekt vom Typ `Val`, das einen Wert vom Typ `int` enthält; oder

– ein Objekt vom Typ `Op`, das ein Operatorsymbol (`OpSym.ADD` oder `OpSym.MUL`) und dessen Argumente enthält, wobei die Argumente selbst wieder beliebige Ausdrucksbäume vom Typ `Node` sein können.

Der von uns bereitgestellte Parser versucht Strings in Ausdrucksbäume dieser Form umzuwandeln, und gibt `None` zurück falls der String keinen gültigen Ausdrucksbaum darstellt. Beispiel:

```
from tree import Node, Var, Val, Op
from parser import parse

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

assert parse("2 * x + 5") == Op(OpSym.ADD, Op(OpSym.MUL, Val(2), Var('x')), Val(5))
assert parse("invalid input") == None
```

Implementieren Sie das zu Beginn beschriebene Programm in folgenden Schritten:

(a) (2 Punkte) Schreiben Sie eine Funktion `node_to_str`, die einen Ausdrucksbaum als Argument nimmt und dessen Darstellung als String zurückgibt. Machen Sie die Klammerung von Operatoren explizit und verwenden Sie genau ein Leerzeichen um Operatoren von Argumenten zu trennen. Beispiel:

```
e = Op(OpSym.ADD, Op(OpSym.MUL, Val(2), Var('x')), Val(5))
assert node_to_str(e)         == '((2 * x) + 5)'
assert node_to_str(Var('x'))  == 'x'
assert node_to_str(Val(2))    == '2'
```

Verwenden Sie hierzu Pattern Matching und *keine* `if`-Verzweigungen.

Hinweis. Die Alternativen von Aufzählungstypen haben ein Feld `value`, welches den Wert der Alternative zurückgibt. Beispiel:

```
assert OpSym.ADD.value == "+"
```

(b) (2 Punkte) Schreiben Sie zum Vergleich eine alternative Implementierung von `node_to_str`, die `if`-Verzweigungen aber *kein* Pattern Matching verwendet. Nennen Sie diese Funktion `node_to_str_if`.

(c) (8 Punkte) Schreiben Sie eine Funktion `optimize_step`, die einen Ausdrucksbaum `e` als Argument nimmt und versucht den Ausdrucksbaum durch Anwenden einer Regel umzuformen. Ist dies möglich, so soll der umgeformte Ausdrucksbaum zurückgegeben werden, ansonsten `None`.

Die Regeln sind wie folgt:

– Ist `e` ein Operator, der auf zwei Zahlen angewendet wird, so soll das Ergebnis der Berechnung zurückgegeben werden. Beispiel:

```
assert optimize_step(Op(OpSym.MUL, Val(5), Val(3))) == Val(15)
```

– Ist `e` ein `+`-Operator, der auf zwei gleiche Argumente angewendet wird, so soll stattdessen eine Multiplikation mit `2` zurückgegeben werden. Beispiel:

```
assert optimize_step(Op(OpSym.ADD, Var('x'), Var('x'))) ==
                     Op(OpSym.MUL, Val(2), Var('x'))
assert optimize_step(Op(OpSym.ADD, Var('x'), Var('y'))) == None
e = Op(OpSym.MUL, Val(3), Var('x'))
assert optimize_step(Op(OpSym.ADD, e, e)) == Op(OpSym.MUL, Val(2), e)
```

Hinweis: Verwenden Sie Pattern Guards!

- Stellt `e` einen Ausdruck der Form `e1 + (e2 + e3)` dar, so soll das Assoziativgesetz angewandt werden, also der Ausdruck `(e1 + e2) + e3` zurückgegeben werden. Analog für `*` statt `+`.
- Trifft keine der vorherigen Regeln zu und `e` ist die Anwendung eines Operators, so soll versucht werden `optimize_step` auf die Argumente des Operators anzuwenden.

  Da es sich bei `optimize_step` um die Anwendung einer einzelnen Regel handelt, soll erst versucht werden das erste Argument umzuformen, und nur wenn das erste Argument nicht umgeformt werden konnte, soll versucht werden das zweite Argument umzuformen. Beispiel:

```
e1 = Op(OpSym.MUL, Val(2), Val(3))
e2 = Op(OpSym.MUL, Val(3), Val(2))
assert optimize_step(Op(OpSym.ADD, e1,     e2)) == Op(OpSym.ADD, Val(6), e2)
assert optimize_step(Op(OpSym.ADD, Val(6), e2)) == Op(OpSym.ADD, Val(6), Val(6))
assert optimize_step(Op(OpSym.ADD, Val(6), Val(6))) == Val(12)
assert optimize_step(Val(12)) == None
```

Versuchen Sie die Regeln in der hier angegebenen Reihenfolge anzuwenden. Verwenden Sie zur Unterscheidung der einzelnen Regeln Pattern Matching und *keine* `if`-Verzweigungen.

(d) (2 Punkte) Schreiben Sie zum Vergleich eine alternative Implementierung von `optimize_step`, die `if`-Verzweigungen aber *kein* Pattern Matching verwendet. Nennen Sie diese Funktion `optimize_step_if`.

(e) (2 Punkte) Schreiben Sie eine Funktion `optimize`, die einen Ausdrucksbaum `e` als Argument nimmt, diesen so lange mit `optimize_step` umformt bis keine Regeln mehr greifen und dann eine Liste aller Zwischenergebnisse inklusive `e` zurückgibt.

Beispiel:

```
assert optimize(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)')
]
```

(f) (2 Punkte) Verwenden Sie die Funktionen `parse`, `optimize` und `node_to_str`,

um eine "Optimizer REPL"[3] zu implementieren, wie im Einleitungsbeispiel beschrieben. Bevor die Benutzereingabe zu einem Baum umgewandelt wird, soll dabei überprüft werden, ob diese gleich `"quit"` ist und in diesem Fall das Programm beendet werden. Ungültige Benutzereingaben sollen mit einer Fehlermeldung ignoriert werden. Die Ausgabe soll dabei *exakt* die Form wie im folgenden Beispiel haben.

Beispiel:

```
$ python3 optimizer.py
> (x + x) + (x + x)
= ((x + x) + (x + x))
= (2 * (x + x))
= (2 * (2 * x))
= ((2 * 2) * x)
= (4 * x)

> 5 ( 23
Syntax error.

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

> quit
Good bye!
```

Aufgabe 7.2 (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 `3.5 h` steht dabei für 3 Stunden 30 Minuten.

Bei Gruppenaufgaben müssen alle Gruppenmitglieder die Abgabe hochladen. Ferner müssen Sie die RZ-Accounts aller Gruppenmitglieder angeben, sodass wir Sie bei der Korrektur zuordnen können und unsere Tutoren die Aufgabe nicht mehrfach korrigieren müssen. Fügen Sie hierzu in der `NOTES.md`-Datei unter der Zeile für den Zeitbedarf eine weitere Zeile hinzu die exakt das folgende Format hat:

```
Gruppe: xy123, xy234, xy345
```

Hierbei stehen `xy123`, `xy234`, und `xy345` für die RZ-Accounts der Gruppenmitglieder.

[3] 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).

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