Westfälische Hochschule

Fachbereich Informatik und Kommunikation
Prof. Dr. Marcel Luis

9. April 2014

Praktikum 3 zu Objektorientierte Programmierung

In dieser und einer späteren Hausaufgabe geht es um das Parsen und Auswerten von Ausdrücken. Die Realisierung der dazu erforderlichen Anwendungsklassen wird Gegenstand einer späteren Hausaufgabe sein. Die *vorherige* Realisierung der Testklassen ist Gegenstand dieser Hausaufgabe.

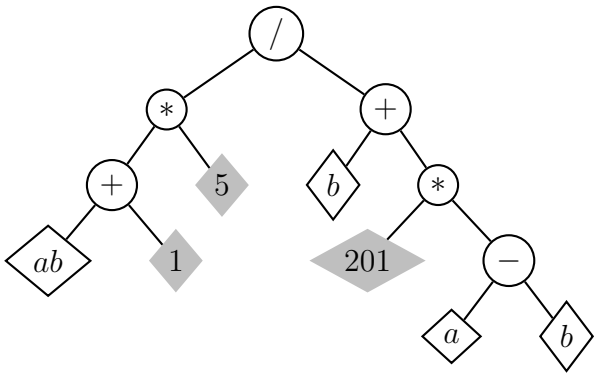
Aufgabe 3.1 (Aufwand ca. 6 Stunden, Abgabe bis 20.04.2014 um 18 Uhr)

Ausdrücke lassen sich auf verschiedene Arten repräsentieren, z. B. textuell in Präfix- oder Infixdarstellung oder strukturell durch Bäume. Die „im Alltag“ übliche Darstellung arithmetischer Ausdrücke, die aus *Konstanten*, *Variablen* und den *vier Grundrechenarten* bestehen, ist die geklammerte Infixdarstellung.

In einer späteren Hausaufgabe werden Sie eine Klasse `Parser` realisieren. Der Parser soll zu einer geklammerten Infixdarstellung eines Ausdrucks dessen Baumrepräsentation liefern. Zur Darstellung der Bäume dienen folgende Klassen:

`Klasse`	Kurzbeschreibung
`Ausdruck`	*abstrakte* Oberklasse aller Ausdrücke
`Konstante`	Konstante als spezieller Ausdruck
`Variable`	Variable als spezieller Ausdruck
`OperatorAusdruck`	spezieller Ausdruck bestehend aus Operatorsymbol und zwei Teilausdrücken

Durch diese Klassen wird z. B. der Ausdruck $(ab+1)*5/(b+201*(a-b))$ so repräsentiert (Kreis-Knoten stehen für Operatorausdrücke, Rauten-Knoten mit weißem Hintergrund für Variablen und Knoten mit grauem Hintergrund für Konstanten):



Damit Sie die Testklassen zum Parsen und Auswerten von Ausdrücken realisieren können, müssen Sie einige grundlegende Informationen über die Anwendungsklassen und deren Methoden haben.

Klasse `Ausdruck`:

- Enthält abstrakte Instanzmethode `int gibWert(Variablenbelegung)`, die den Wert dieses Ausdrucks basierend auf der Variablenbelegung liefert.

Die Klassen `Konstante`, `Variable` und `OperatorAusdruck` sind Unterklassen von `Ausdruck`.

Klasse `Konstante`:

- Enthält Konstruktor `Konstante(int)`, durch den ein konstanter Ausdruck für den angegebenen Wert erzeugt wird.

Klasse `Variable`:

- Enthält Konstruktor `Variable(String)`, durch den eine Variable mit dem angegebenen Namen erzeugt wird.

Klasse `OperatorAusdruck`:

- Enthält Konstruktor `OperatorAusdruck(Ausdruck, char, Ausdruck)`, durch den ein arithmetischer Ausdruck mit den angegebenen Teilausdrücken und dem Operatorsymbol erzeugt wird.

Klasse `Variablenbelegung`:

- Enthält Konstruktor `Variablenbelegung()`, durch den eine Variablenbelegung erzeugt wird, in der zunächst keiner Variablen ein Wert zugeordnet ist.
- Enthält Instanzmethode `void belege(String, int)`, durch die einer Variablen (1. Parameter) ein Wert (2. Parameter) zugeordnet wird. Ein evtl. vorhandener alter Wert wird dabei überschrieben.
- Enthält Instanzmethode `int gibWert(String)`, die den Wert liefert, der der angegebenen Variable zugeordnet ist.

Wozu benötigt man diese Klasse? Stellen Sie sich vor, Sie sollen den Ausdruck $a+2+x1$ auswerten. Sie werden schnell feststellen, dass dies nicht möglich ist, ohne die Werte von a und $x1$ zu kennen. Ist die Belegung dieser Variablen z. B. $a \mapsto 5$ und $x1 \mapsto -20$, dann lässt sich der Ausdruck zu -13 auswerten.

Genau hierum geht es bei der Klasse `Variablenbelegung`. Ein Objekt dieser Klasse repräsentiert die Beziehung zwischen (vielen) Variablen und ihren Werten.

Die obige beispielhafte Belegung von Variablen a und $x1$ wird wie folgt erzeugt:

1. Objekt der Klasse `Variablenbelegung` erzeugen.
2. Auf dieses Objekt `belege("a", 5)` anwenden.
3. Auf dasselbe Objekte `belege("x1", -20)` anwenden.

Eine Frage, die in diesem Zusammenhang häufig gestellt wird: Warum wird der Wert einer Variablen nicht schon im Objekt der Klasse `Variable` verwaltet? Weil ein Objekt dieser Klasse lediglich das *Auftreten* der Variablen innerhalb eines Ausdrucks repräsentiert (also die syntaktische Seite der Variablen).

Klasse `Parser`:

- Enthält Instanzmethode `Ausdruck parse(String)`, die die Textdarstellung (gewöhnliche geklammerte Infixdarstellung) eines Ausdrucks parst und ein entsprechendes `Ausdruck`-Objekt zurückgibt. Beliebig viele Leerzeichen (einschließlich 0 Leerzeichen!) zwischen den Komponenten des Ausdrucks sind zulässig.

Realisieren Sie nun in dieser Hausaufgabe basierend auf JUnit die Testklassen `ParserTest` (darin Testmethode `testParse`) und `AusdruckTest` (darin Testmethode `testGibWert`). Die Klassen sollen im Paket `ausdruck` angelegt werden.

Die Testmethoden einer Testklasse stellen eine Spezifikation des gewünschten Verhaltens einer Klasse anhand von Beispielen (Testmustern) dar. Überlegen Sie sich also, durch welche Beispiele Sie das Parsen und Auswerten von Ausdrücken treffend spezifizieren können und realisieren Sie die entsprechenden Testmuster in den Testmethoden.

Für das Parsen sieht eine solche Realisierung z. B. so aus:

```
assertEquals(sollAusdruck, parser.parse("(a + 1) * 5"));
```

`sollAusdruck` ist dabei der Ausdruck, den Sie als Ergebnis des Aufrufs

```
parser.parse("(a + 1) * 5")
```

erwarten. Den Sollausdruck können Sie mit Hilfe der Konstruktoren erzeugen:

```
Ausdruck sollAusdruck
        = new OperatorAusdruck(
                  new OperatorAusdruck(
                          new Variable("a"), '+', new Konstante(1)),
                  '*',
                  new Konstante(5));
```

Bleibt die Frage, wie JUnit wissen soll, wann zwei Ausdrücke gleich (!) sind. Dafür müssen Sie in den Klassen `Konstante`, `Variable` und `OperatorAusdruck` die Methode `equals` überschreiben. So kann jedes Objekt dieser Klassen entscheiden, ob es gleich einem beliebigen anderen Objekt ist. Hierauf basiert das Verhalten der Methode `void assertEquals(Object, Object)` für zwei Objekte.

Das Ziel dieser Hausaufgabe haben Sie erreicht, wenn Sie die beiden Testklassen mit sinnvollen Testmustern (mindestens 6 je Testmethode) realisiert haben und sich diese Klassen fehlerfrei compilieren lassen. Dazu müssen Sie selbstverständlich die „Skelette“ der Anwendungsklassen mit den oben angegebenen Konstruktoren und Methoden anlegen, und für manche Methoden müssen Sie irgendwelche `return`-Anweisungen implementieren, damit sich die Klassen compilieren lassen. Legen Sie alle Klassen im Paket `ausdruck` an.

Außerdem sollen die Klassen `Konstante`, `Variable` und `OperatorAusdruck` sinnvolle Implementierungen der `equals`-Methode enthalten. Die *Algorithmen* zum Parsen und Auswerten und die tatsächliche Verwaltung von Variablenbelegungen sollen Sie noch nicht realisieren. Die Checkstyle-Prüfung der drei Klassen wird einen Hinweis zu einer Methode `hashCode` enthalten, den Sie zunächst ignorieren können.

Wenn Sie die Testklassen ausführen, also die Tests *durchführen*, werden diese selbstverständlich fehlschlagen. Das ist zu diesem Zeitpunkt unvermeidlich. Bedenken Sie: Sie spezifizieren mit dieser Hausaufgabe nur die *Anforderungen* an den Parser und die *Anforderungen* an das Auswerten von Ausdrücken. Diese Anforderungen zu erfüllen, wird Gegenstand einer späteren Hausaufgabe sein.

Hinweise

- Achten Sie darauf, dass in den Eigenschaften Ihres NetBeans-Projekts die Zeichenkodierung UTF-8 eingestellt ist. Ist dies nicht der Fall, kann es bei der automatischen Auswertung Ihrer Lösung zu Fehlern beim Compilieren kommen. Die Lösung wird dann mit einer Erfolgsquote von 0% gewertet.

- Ich rate Ihnen dringend, der Versuchung zu widerstehen, *erst* das Parsen und Auswerten zu realisieren, und *dann* den Test. Denn:

    1. Sie bringen sich um die Erfahrung, wie sinnvoll es ist, erst die Anforderungen an eine Anwendung möglichst präzise zu spezifizieren (nichts Anderes ist ein programmierter JUnit-Test), und danach die Anwendung zu realisieren.

    2. Sie vergeuden wertvolle Zeit, denn Sie erhalten später noch Hinweise, wie sich das Parsen systematisch realisieren lässt.

- Dokumentieren Sie wie üblich alle Klassen und Methoden. Erstellen Sie je ein `zip`-Archiv

des Quellordners `ausdruck` unter `src` und des gleichnamigen Ordners unter `test` und laden Sie beide zu Moodle hoch.