

Software Entwicklung 1

Annette Bieniusa / Arnd Poetzsch-Heffter

AG Softech
FB Informatik
TU Kaiserslautern

Fallstudie: Arithmetische Ausdrücke

Überblick

- Rückblick und Wiederholung
 - Subtyping und Vererbung
 - Statische Methoden
 - Rekursive Datenstrukturen (Bäume)
 - Grammatiken
 - Parsen durch rekursiven Abstieg
- Design Pattern: Factory
- Eliminierung von Linksrekursion in Grammatiken
- Testen von Exceptions

Rekursive Klassen

Eine Klasse ist **rekursiv**, falls sie ein Attribut der gleichen Klasse, einer Superklasse oder eines implementierten Interfaces besitzt.

Typische Beispiele:

- Knoten von verketteten Listen
- Baumstrukturen

Arithmetische Ausdrücke

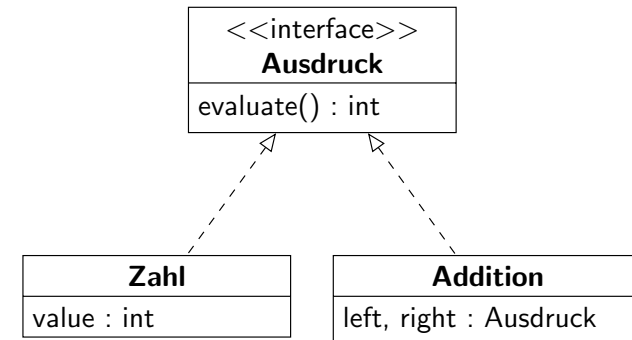
Ein arithmetischer Ausdruck ist entweder

- eine Konstante (eine ganze Zahl)
Beispiele: 0, 51, -42
- eine Summe von zwei arithmetischen Ausdrücken
Beispiele: 3+4, 17+4, 17+ (2+2)

Typische Operationen auf arithmetischen Ausdrücken

- Berechnen des Werts

Arithmetische Ausdrücke: Klassendiagramm



Methodenentwurf: Auswertung von Konstanten

- Der Wert einer Konstanten ist die Konstante selbst.

```
assertEquals(42, new Zahl(42).evaluate());
```

- Implementierung

```
public class Zahl implements Ausdruck {
    // Felder und Konstruktor ...

    public int evaluate() {
        return this.value;
    }
}
```

Methodenentwurf: Auswertung von Summen

- Der Wert eines **Addition**-Ausdrucks ist die Summe der Werte seiner Teilausdrücke.

```
assertEquals(21, new Addition(new Zahl(17), new Zahl(4)).evaluate());
assertEquals(21, new Addition(new Zahl(17),
    new Addition(new Zahl(2),
        new Zahl(2))).evaluate());
```

- Implementierung

```
public class Addition implements Ausdruck {
    // Felder und Konstruktoren ...

    public int evaluate() {
        return left.evaluate() + right.evaluate();
    }
}
```

Erweiterung: Neue Arten von Ausdrücken

Ein arithmetischer Ausdruck ist entweder

- eine Konstante (eine natürliche Zahl)
Bsp.: 0, 51, 42
- eine Summe von zwei arithmetischen Ausdrücken
Bsp.: 3+4, 17+4, 17+ (2+2)
- ein Produkt von zwei arithmetischen Ausdrücken
Bsp.: 3*4, 2 * (17+4), (2 * 3) * 4

Erweiterung: Neue Operation hinzufügen

Ein arithmetischer Ausdruck kann

- mit `evaluate()` seinen Wert berechnen;
- mit `size()` seine Größe berechnen.

Die Größe eines arithmetischen Ausdrucks ist die Anzahl seiner Operatoren und Konstanten.

Arithmetische Ausdrücke: Erweiterung

Zum Hinzufügen einer neuen Art von Ausdruck muss nur eine neue Klasse definiert werden, die das Interface `Ausdruck` implementiert.

```
class Multiplikation implements Ausdruck {
    private Ausdruck left;
    private Ausdruck right;

    Multiplikation(Ausdruck left, Ausdruck right) {
        this.left = left;
        this.right = right;
    }

    public int evaluate() {
        return left.evaluate() * right.evaluate();
    }

    public String toString() {
        return "(" + left + "*" + right + ")";
    }
}
```

- Interface anpassen

```
public interface Ausdruck {
    int eval();
    int size();
}
```

- In jeder Klasse, die das Interface implementiert, muss die neue Methode `size()` hinzugefügt werden

Nachteil

- Die Änderung ist nicht lokal.
- Es müssen ggf. viele Klassen angepasst werden.

Methodenentwurf: `size()` von Konstanten

- Die Größe einer Konstanten ist 1.

```
assertEquals(1, new Zahl(42).size());
```

- Implementierung

```
public class Zahl implements Ausdruck {
    ...
    public int size() {
        return 1;
    }
}
```

Erzeugen von Beispiel-Ausdrücken

Mit Java Konstruktoren

```
@Test
public void testEval() {
    // (42 + (5 * 4)) * (2 + 1)
    Ausdruck e = new Produkt(new Addition(new Zahl(42),
        new Produkt(new Zahl(5), new Zahl(4))),
        new Add(new Zahl(2), new Zahl(1)));
    assertEquals(186, e.evaluate());
}
```

Java's Syntax für Konstruktoraufrufe ist für das **Erstellen verschachtelter Strukturen** sehr unleserlich.

Methodenentwurf `size()` von Summen

- Die Größe eines `Addition`-Ausdrucks ist die Summe der Größen seiner Teilausdrücke plus 1 für die Addition.

```
assertEquals(3, new Addition(new Zahl(17), new Zahl(4)).size());
assertEquals(5, new Addition(new Zahl(17),
    new Addition(new Zahl(2),
        new Zahl(2))).size());
```

- Implementierung

```
public class Addition implements Ausdruck
    ...
    public int size() {
        return 1 + left.size() + right.size();
    }
}
```

- Analog bei Produkten!

Erzeugen von Beispiel-Ausdrücken

Mit statischen Factory-Methoden

```
@Test
public void testEvalStatic() {
    // (42 + (5 * 4)) * (2 + 1)
    Ausdruck e = mult(add(cnst(42), mult(cnst(5), cnst(4))),
        add(cnst(2), cnst(1)));
    assertEquals(186, e.evaluate());
}
```

- Die statischen Methoden `mult` und `add` sind **statische Methoden** aus einer Klasse `AusdruckFactory`.

Parsen

Aufgabenstellung Parsen

Gesucht ist ein Parser-Objekt `p` für einen String `input` mit Methode

```
Ausdruck parse();
```

so dass folgendes gilt:

- 1 Falls `input` einen korrekt geformten Ausdruck enthält, so liefert `parse()` ein Objekt von Type `Ausdruck`, das der Eingabe entspricht.
- 2 Falls `input` keinen korrekt geformten Ausdruck enthält, so liefert `parse()` eine Exception als Fehlermeldung.
- 3 Für alle Objekte `e` ungleich `null` von Typ `Ausdruck` soll gelten, dass `p.parse(e.toString()).toString()` den gleichen String liefert wie `e.toString()`. Insbesondere wird keine Exception ausgelöst.

Beispiele für `toString()`

```
public void test() {
    checkParseToString(cnst(45));
    checkParseToString(add(cnst(3), cnst(4)));
    checkParseToString(mult(cnst(3), cnst(4)));
    checkParseToString(add(mult(cnst(2), cnst(3)), cnst(4)));
}

private void checkParseToString(Ausdruck e) {
    String r = e.toString();
    Parser p = new Parser(r);
    Ausdruck e_parsed = p.parse();
    assertEquals(e.toString(), e_parsed.toString());
}
```

Lexeme: Bestandteile eines Ausdrucks

- 1 Konstante: Folge von Ziffern
- 2 Klammer auf
- 3 Klammer zu
- 4 Pluszeichen
- 5 Multiplikationszeichen
- 6 Trennsymbole: Leerzeichen, Tabulatoren, etc (**whitespace**) (hier nicht betrachtet)

Die Zeichenfolgen nach 1-5 heißen **Lexeme**.

Zwischen zwei Lexemen dürfen beliebig viele Trennsymbole eingefügt werden.

Grammatik für Ausdrücke

$$\begin{aligned} \text{Ausdruck} & ::= \text{Ausdruck} + \text{Term} \\ & \quad | \text{Term} \\ \text{Term} & ::= \text{Term} * \text{Factor} \\ & \quad | \text{Factor} \\ \text{Factor} & ::= \text{Zahl} \\ & \quad | (\text{Ausdruck}) \end{aligned}$$

- Für diese Grammatik gibt es zu jedem String höchstens eine Ableitung.
- Die Ableitung $\text{Ausdruck} \Rightarrow \dots \Rightarrow 1 + 2 * 3$ entspricht $1 + (2 * 3)$

Wunschzettelaktion

Welches Thema wünschen Sie sich im neuen Jahr?

- Muss Bezug zum Programmieren oder Software-Engineering haben
- Bitte notieren Sie Ihren Wunsch auf einem Zettel!

Parsen durch rekursiven Abstieg

Parsen durch rekursiven Abstieg: Idee

- Jedes Nichtterminalsymbol N wird durch eine Methode `parseN` repräsentiert.
- Die Methode zum Nichtterminalsymbol N liest aus der Eingabe einen String, der aus N abgeleitet werden kann.
- Die Definition der Methode ergibt sich aus den Produktionen für N .
 - Anhand des vorliegenden Lexems wird eine Alternative der Produktion ausgewählt.
 - Für die Symbole auf der rechten Seite wird von rechts nach links Code generiert.
 - Für ein Nichtterminalsymbol N' wird `parseN'` aufgerufen.
 - Für ein Terminalsymbol t wird `tryParseT()` aufgerufen.
- Bemerkung: Die Methoden rufen sich gegenseitig rekursiv auf!

Beispiel: Methode für Factor

$$\text{Factor} ::= \text{Zahl} \mid (\text{Ausdruck})$$

```
Ausdruck parseFactor() throws ParseException {
    Zahl z = tryParseZahl();
    if (z != null) {
        return z;
    }
    if (tryParseKlammerAuf() != null) {
        Ausdruck e = parseAusdruck();
        if (tryParseKlammerZu() != null) {
            return e;
        } else {
            throw new ParseException("Klammer zu erwartet", pos);
        }
    }
    throw new ParseException("Kann keinen Factor parsen", pos);
}
```

Linksrekursion

Problem: Methode für Ausdruck

- Verfahren funktioniert gut für Factor, weil jede Alternative mit einem anderen Lexem beginnt.
- Bei den anderen Nichtterminalen ist das nicht der Fall.
- Betrachte das Beispiel Ausdruck

$$\text{Ausdruck} ::= \text{Ausdruck} + \text{Term} \mid \text{Term}$$

```
public Ausdruck parseAusdruck() {
    Ausdruck left = parseAusdruck();
    // oops, rekursiver Aufruf terminiert nicht!!!
    // ...
}
```

- Um dieses Problem zu vermeiden, muss die Grammatik umstrukturiert werden!

Elimination von Linksrekursion

Die Produktionen für Ausdruck und Term sind **linksrekursiv**, d.h., das gleiche Nichtterminalsymbol taucht direkt am Anfang der rechten Seite einer Regel auf.

$$\begin{aligned} \text{Ausdruck} & ::= \text{Ausdruck} + \text{Term} \mid \text{Term} \\ \text{Term} & ::= \text{Term} * \text{Factor} \mid \text{Factor} \end{aligned}$$

Diese Produktionen können umgeformt werden, so dass sie nicht mehr linksrekursiv sind, aber dass die gleiche Sprache erkannt wird.

$$\begin{aligned} \text{Ausdruck} & ::= \text{Term Ausdruck1} \\ \text{Ausdruck1} & ::= \varepsilon \mid + \text{Term Ausdruck1} \\ \text{Term} & ::= \text{Factor Term1} \\ \text{Term1} & ::= \varepsilon \mid * \text{Factor Term1} \end{aligned}$$

Unterscheiden der Alternativen durch Lookahead

Ausdruck ::= Term Ausdruck1
 Ausdruck1 ::= ϵ | + Term Ausdruck1
 Term ::= Factor Term1
 Term1 ::= ϵ | * Factor Term1
 Factor ::= Zahl | (Ausdruck)

Lookahead

Erkennen der Regel-Alternativen durch Lookahead

- Ausdruck, Term: nur eine Alternative
- Ausdruck1: durch Testen des folgenden Lexems (Lookahead)
 - falls + in der Eingabe, dann zweite Regel
 - sonst, erste (leere) Regel
- Term1: durch Testen des folgenden Lexems (Lookahead)
 - falls * in der Eingabe, dann zweite Regel
 - sonst, erste (leere) Regel
- Factor: Testen des Lexems (Zahl oder Klammer auf)

Beispiel: Methode für Term1

```

Ausdruck parseTerm1(Ausdruck left) throws ParseException {
    if (tryParseStar() != null) {
        Ausdruck right = parseFactor();
        return parseTerm1(new Multiplikation(left, right));
    }
    return left;
}
  
```

- `parseTerm1` erhält beim Aufruf den linken Faktor als Argument.
- Dadurch wird * links-assoziativ.
- `parseAusdruck1` wird analog implementiert.

Erinnerung

- Probeklausur: 12.01.2016!
- Anmeldung über STATS
- Feedback zur Vorlesung über die VLU!
- Übungsblatt 9 ist bereits online

Frohe Weihnachten und einen guten Start in 2016!

