Programmieren in Java

Vorlesung 07: Parsen

Prof. Dr. Peter Thiemann

Albert-Ludwigs-Universität Freiburg, Germany

SS 2015

Inhalt

Vorlesungsüberblick

-

Erinnerung

Ein arithmetischer Ausdruck hat eine der folgenden Formen:

- eine Variable
  Beispiele: x, index
- eine Konstante (eine ganze Zahl)
  Beispiele: 0, 51, -42
- eine Summe von zwei arithmetischen Ausdrücken
  Beispiele: 3+4, 17+4, 17+ (2+2)
- ein Produkt von zwei arithmetischen Ausdrücken
  Beispiele: 3∗4, $2 * (17+4)$, $(2 * 3) * 4$

Erstellen von Ausdrücken

Mit Hilfe von Konstruktoren

- eine Variable
  Beispiele: **new** Var("x"), **new** Var("index")
- eine Konstante (eine ganze Zahl)
  Beispiele: **new** Const(0), **new** Const(51), **new** Const(−42)
- eine Summe von zwei arithmetischen Ausdrücken
  Beispiele: **new** Add(**new** Const(3), **new** Const(4)),
  new Add(**new** Const(17), **new** Add(**new** Const(2), **new** Const(2))
- ein Produkt von zwei arithmetischen Ausdrücken
  Beispiele: **new** Product(**new** Const(3), **new** Const(4)),
  new Product(**new** Product(**new** Const(2), **new** Const(3)), **new** Const(4))

Erstellen von Ausdrücken

Mit Hilfe von statischen Factory Methoden

- eine Variable
  Beispiele: var("x"), var("index")
- eine Konstante (eine ganze Zahl)
  Beispiele: cnst(0), cnst(51), cnst(−42)
- eine Summe von zwei arithmetischen Ausdrücken
  Beispiele: add(cnst(3), cnst(4)), add(cnst(17), add(cnst(2), cnst(2))
- ein Produkt von zwei arithmetischen Ausdrücken
  Beispiele: product(cnst(3), cnst(4)), product(product(cnst(2), cnst(3)), cnst(4))

Heute: Erstellen von Ausdrücken durch **Parsen**

D.h. Einlesen von einem String oder aus einer Datei

- eine Variable
  Beispiele: parse("x"), parse("index")
- eine Konstante (eine ganze Zahl)
  Beispiele: parse("0"), parse("51"), cnst(−42)
- eine Summe von zwei arithmetischen Ausdrücken
  Beispiele: parse("3+4"), parse("17+(2+2)")
- ein Produkt von zwei arithmetischen Ausdrücken
  Beispiele: parse("3 *4"), parse("2 * 3 * 4")

Aufgabenstellung Parsen

Gesucht

Statische Methode

```
1 IExpr parse(String input);
```

so dass folgendes gilt:

1. Falls input einen korrekt geformten Ausdruck enthält, so liefert parse (input) ein Objekt von Type IExp, das der Eingabe entspricht.
2. Falls input keinen korrekt geformten Ausdruck enthält, so liefert parse (input) eine Exception als Fehlermeldung.
3. Für alle Objekte e0 ungleich **null** von Typ IExpr soll gelten, dass e0.equals(parse (e0.toString())) == **true** ist. Insbesondere wird keine Exception ausgeloest.

Definition von toString()

```
1  class Var implements IExpr {
2    public String toString() {
3     return name;
4    }
5  }
6  class Const implements IExpr {
7    public String toString() {
8      return value + "";
9    }
10 }
11 class Add implements IExpr {
12   public String toString() {
13    return "(" + left.toString() + " + " + right.toString() + ")";
14   }
15 }
16 class Product implements IExpr {
17   public String toString() {
18    return "(" + left.toString() + " * " + right.toString() + ")";
19   }
20 }
```

Beispiele für toString()

```
1   public void test() {
2     checkToString("x", var("x"));
3     checkToString("45", cnst(45));
4     checkToString("(3 + 4)", add(cnst(3), cnst(4)));
5     checkToString("(3 * 4)", product(cnst(3), cnst(4)));
6     checkToString("((2 * 3) + 4)", add(product(cnst(2),cnst(3)), cnst(4)));
7   }
8
9   private void checkToString(String x, IExpr e) {
10    String r = e.toString();
11    System.out.println("" + r + "");
12    assertEquals(x, r);
13  }
```

Beispiele für parse()

Umkehrung von toString()

```
1   public void testParse() {
2     checkParse("x", var("x"));
3     checkParse("45", cnst(45));
4     checkParse("(3 + 4)", add(cnst(3), cnst(4)));
5     checkParse("(3 * 4)", product(cnst(3), cnst(4)));
6     checkParse("((2 * 3) + 4)", add(product(cnst(2),cnst(3)), cnst(4)));
7   }
8
9   private void checkParse(String input, IExpr expected) {
10    IExpr result = parse(input);
11    assertEquals(expected, result);
12  }
```

Lexeme: Bestandteile eines Ausdrucks

1. Variable: String bestehend aus mehr als einem Buchstaben
   "[A−Za−z]+"
2. Konstante: String bestehend aus mehr als einer Ziffer "\\d+"
3. Klammer auf "\\("
4. Klammer zu "\\)"
5. Pluszeichen "\\+"
6. Multiplikationszeichen "*"
7. Trennsymbole: Leerzeichen, Tabulatoren, etc (**whitespace**)

Lexeme: Bestandteile eines Ausdrucks

1. Variable: String bestehend aus mehr als einem Buchstaben
   "[A−Za−z]+"
2. Konstante: String bestehend aus mehr als einer Ziffer "\\d+"
3. Klammer auf "\\("
4. Klammer zu "\\)"
5. Pluszeichen "\\+"
6. Multiplikationszeichen "*"
7. Trennsymbole: Leerzeichen, Tabulatoren, etc (**whitespace**)

Lexeme

Die Zeichenfolgen nach 1-6 heißen **Lexeme**. Zwischen zwei Lexemen dürfen beliebig viele Trennsymbole eingefügt werden. Traditionell werden Lexeme durch reguläre Ausdrücke definiert (rechte Spalte).

Scanner: Zerlegen eines Ausdrucks in Lexeme

```
1  public interface IScanner {
2    /**
3     * Test if next lexeme in input is matched by regex.
4     * @param regex a regular expression
5     * @return true if input starts with regex.
6     */
7    boolean lookingAt(String regex);
8    /**
9     * If scanner is looking at regex, return string matched by regex and
10    * advance to next lexeme skipping over whitespace.
11    * @param regex a regular expression
12    * @return matched string if input starts with regex. Otherwise return null.
13    */
14   String getLexeme(String regex);
15 }
```

StringScanner: Scanner mit Eingabestring

Definiere Klasse StringScanner mit Feld String input.

```
1  public boolean lookingAt(String regex) {
2    Matcher m = Pattern.compile(regex).matcher(input);
3    return m.lookingAt();
4  }
```

Pattern java.util.regex.Pattern.compile(String regex)

Compiles the given regular expression into a pattern.

- Parameters: regex The expression to be compiled
- Returns: the given regular expression compiled into a pattern
- Throws: PatternSyntaxException - If the expression's syntax is invalid

Pattern und Matcher

Matcher java.util.regex.Pattern.matcher(String input)

Creates a matcher that will match the given input against this pattern.

- Parameters: input The character sequence to be matched
- Returns: A new matcher for this pattern

boolean java.util.regex.Matcher.lookingAt()

Attempts to match the input sequence, starting at the beginning of the region, against the pattern. This method always starts at the beginning of the region. It does **not** require that the entire region be matched.
If the match succeeds then more information can be obtained via the start, end, and group methods.

- Returns: true if, and only if, a prefix of the input sequence matches this matcher's pattern

StringScanner.getLexeme()

```
1  public String getLexeme(String regexp) {
2    Matcher m = Pattern.compile(regexp).matcher(input);
3    String r = null;
4    if (m.lookingAt()) {
5      r = input.substring(0, m.end());
6      input = input.substring(m.end()).trim();
7    }
8    return r;
9  }
```

- substring(begin, end) liefert den Substring ab begin bis (exklusive) end
- end() liefert den Index direkt hinter dem Match
- trim() entfernt Leerzeichen etc zu Beginn eines Strings

Zwischenstand

Scanner

Liefert Zerlegung der Eingabe in Folge von Lexemen und entfernt whitespace. Klassifiziert die Lexeme in **Token**.
Beispiele:

- `42*17+4` $\Rightarrow$ `42|*|17|+|4` $\Rightarrow$ `NUM|*|NUM|+|NUM`
- `(index - 1) * span` $\Rightarrow$ `(|index|-|1|)|*|span` $\Rightarrow$ `(|VAR|-|NUM|)|*|VAR`

Zwischenstand

Scanner

Liefert Zerlegung der Eingabe in Folge von Lexemen und entfernt whitespace. Klassifiziert die Lexeme in **Token**.
Beispiele:

- `42*17+4` $\Rightarrow$ `42|*|17|+|4` $\Rightarrow$ `NUM|*|NUM|+|NUM`
- `(index - 1) * span` $\Rightarrow$ `(|index|-|1|)|*|span` $\Rightarrow$ `(|VAR|-|NUM|)|*|VAR`

Parser

Rekonstruiert aus einer Tokenfolge einen Ausdruck, falls möglich.
Das heisst, der Parser **erkennt die Sprache der Ausdrücke.**

Spezifikation einer Sprache durch BNF

Ein arithmetischer Ausdruck hat eine der folgenden Formen:

- eine Variable
- eine Konstante (eine ganze Zahl)
- eine Summe von zwei arithmetischen Ausdrücken
- ein Produkt von zwei arithmetischen Ausdrücken

Spezifikation einer Sprache durch BNF

Ein arithmetischer Ausdruck hat eine der folgenden Formen:

- eine Variable
- eine Konstante (eine ganze Zahl)
- eine Summe von zwei arithmetischen Ausdrücken
- ein Produkt von zwei arithmetischen Ausdrücken

Sprachdefinition durch BNF

Analog zur Spezifikation von arithmetischen Ausdrücken

$\langle Expr \rangle$	::=	$\langle VAR \rangle$	Variable
	|	$\langle NUM \rangle$	Konstante
	|	$\langle Expr \rangle$ + $\langle Expr \rangle$	Summe
	|	$\langle Expr \rangle$ * $\langle Expr \rangle$	Produkt
	|	($\langle Expr \rangle$)	Klammern (neu)

BNF (Backus-Normalform)

$\langle Expr \rangle$	::=	$\langle VAR \rangle$	Variable
	|	$\langle NUM \rangle$	Konstante
	|	$\langle Expr \rangle$ + $\langle Expr \rangle$	Summe
	|	$\langle Expr \rangle$ * $\langle Expr \rangle$	Produkt
	|	($\langle Expr \rangle$)	Klammern (neu)

BNF Bestandteile

- Variable (Nichtterminalsymbole): $\langle Expr \rangle$, $\langle VAR \rangle$, $\langle NUM \rangle$
  - stehen jeweils für (eine Menge von) Strings
  - $\langle VAR \rangle$ und $\langle NUM \rangle$ stehen für die entsprechenden Lexeme
- Terminalsymbole: +, *, (,)
  stehen jeweils für sich selbst
- Metasymbole ::= und | definieren **Ersetzungsregeln** (Produktionen)

BNF Ersetzungsregeln

$$\begin{aligned} \langle Expr \rangle \quad &::= \quad \langle VAR \rangle \mid \langle NUM \rangle \mid (\ \langle Expr \rangle\ ) \\ &\mid \quad \langle Expr \rangle + \langle Expr \rangle \mid \langle Expr \rangle * \langle Expr \rangle \end{aligned}$$

Ersetzungsregeln

- Links von ::= steht eine Variable
- Rechts von ::= stehen alternative Ersetzungen getrennt durch |
- In jedem Schritt wird eine Variable durch eine ihrer rechten Seiten ersetzt (**Ableitungsschritt** $\Rightarrow$) oder durch ein passendes Lexem
- Beispiele
  - $\langle Expr \rangle \Rightarrow \langle VAR \rangle \Rightarrow$ `index`
  - $\begin{aligned} \langle Expr \rangle &\Rightarrow \langle Expr \rangle + \langle Expr \rangle \\ &\Rightarrow \langle VAR \rangle + \langle Expr \rangle \\ &\Rightarrow \texttt{x} + \langle Expr \rangle \\ &\Rightarrow \texttt{x} + \langle NUM \rangle \\ &\Rightarrow \texttt{x} + 1 \end{aligned}$

Sprache einer BNF

- In jeder BNF ist eine Variable das **Startsymbol**
  Beispiel Ausdrücke: $\langle Expr \rangle$
- Zur **Sprache einer BNF** gehört jeder String, der
  - aus dem Startsymbol abgeleitet werden kann und
  - vollständig aus Terminalsymbolen besteht.

  Beispiel Ausdrücke: `42`, `index`, `x+1`, `2*x+1`, `2*(x+1)`

Problemstellung Parsen

Gegeben

- BNF
- Eingabestring

Gesucht

- Ableitung vom Startsymbol der BNF zum Eingabestring (falls existiert)

Darstellung der Ableitung = IExpr-Objekt

$\langle Expr \rangle$	::=	$\langle VAR \rangle$	Variable	`var(v)`
	|	$\langle NUM \rangle$	Konstante	`cnst(n)`
	|	$\langle Expr \rangle$ + $\langle Expr \rangle$	Summe	`add(`e_1, e_2`)`
	|	$\langle Expr \rangle$ * $\langle Expr \rangle$	Produkt	`product(`e_1, e_2`)`
	|	($\langle Expr \rangle$)	Klammern (neu)	`e`

- Jede Produktion entspricht einem IExpr-Konstruktor (Ausnahme: Klammer)
- Jede vollständige Ableitung entspricht IExpr-Objekt
- Beispiel:

$\langle Expr \rangle \Rightarrow \langle Expr \rangle + \langle Expr \rangle$	`add(`e_1, e_2`)`
$\Rightarrow \langle VAR \rangle + \langle Expr \rangle$	`add(var(v),` e_2`)`
$\Rightarrow$ x $+ \langle Expr \rangle$	`add(var("x"),` e_2`)`
$\Rightarrow$ x $+ \langle NUM \rangle$	`add(var("x"), cnst(n))`
$\Rightarrow$ x + 1	`add(var("x"), cnst(1))`

Problem: Ausdrucks-BNF ist nicht eindeutig

Betrachte 1+2*3

$$
\begin{aligned}
& \langle Expr \rangle \\
\Rightarrow\ & \langle Expr \rangle + \langle Expr \rangle \\
\Rightarrow\ & \langle NUM \rangle + \langle Expr \rangle \\
\Rightarrow\ & 1 + \langle Expr \rangle \\
\Rightarrow\ & 1 + \langle Expr \rangle * \langle Expr \rangle \\
\Rightarrow\ & 1 + \langle NUM \rangle * \langle Expr \rangle \\
\Rightarrow\ & 1 + 2 * \langle Expr \rangle \\
\Rightarrow\ & 1 + 2 * \langle NUM \rangle \\
\Rightarrow\ & 1 + 2 * 3
\end{aligned}
$$

$$
\begin{aligned}
& \langle Expr \rangle \\
\Rightarrow\ & \langle Expr \rangle * \langle Expr \rangle \\
\Rightarrow\ & \langle Expr \rangle + \langle Expr \rangle * \langle Expr \rangle \\
\Rightarrow\ & \langle NUM \rangle + \langle Expr \rangle * \langle Expr \rangle \\
\Rightarrow\ & 1 + \langle Expr \rangle * \langle Expr \rangle \\
\Rightarrow\ & 1 + \langle NUM \rangle * \langle Expr \rangle \\
\Rightarrow\ & 1 + 2 * \langle Expr \rangle \\
\Rightarrow\ & 1 + 2 * \langle NUM \rangle \\
\Rightarrow\ & 1 + 2 * 3
\end{aligned}
$$

```
1 + (2 * 3)
add(cnst(1),
    prd(cnst(2),
        cnst(3)))
```

```
(1 + 2) * 3
prd(add(cnst(1),
        cnst(2)),
    cnst(3))
```

Lösung: Eindeutige BNF für Ausdrücke

$$
\begin{array}{lcl}
\langle \mathit{Expr} \rangle & ::= & \langle \mathit{Expr} \rangle + \langle \mathit{Term} \rangle \\
& | & \langle \mathit{Term} \rangle \\
\langle \mathit{Term} \rangle & ::= & \langle \mathit{Term} \rangle * \langle \mathit{Factor} \rangle \\
& | & \langle \mathit{Factor} \rangle \\
\langle \mathit{Factor} \rangle & ::= & \langle \mathit{VAR} \rangle \\
& | & \langle \mathit{NUM} \rangle \\
& | & (\ \langle \mathit{Expr} \rangle\ )
\end{array}
$$

- Für diese BNF gibt es zu jedem String höchstens eine Ableitung
- Punktrechnung vor Strichrechnung ist “eingebaut”
- Die Ableitung $\langle \mathit{Expr} \rangle \Rightarrow \cdots \Rightarrow 1 + 2 * 3$
  entspricht `1 + (2 * 3)`

BNF $\rightarrow$ Recursive Descent Parser

Idee des Parsers

- Jede Variable N wird durch eine Methode parseN repräsentiert.
- Die Methode zu Variable 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 (Tokens) wird eine Alternative der Produktion ausgewählt.
  - Für die Symbole auf der rechten Seite wird von rechts nach links Code generiert.
  - Für eine Variable N' wird parseN' aufgerufen.
  - Für ein Terminalsymbol t wird getLexeme(t) aufgerufen.
- Bemerkung: die Methoden rufen sich gegenseitig rekursiv auf!

Beispiel: Methode für $\langle Factor \rangle$

$$\langle Factor \rangle ::= \langle VAR \rangle \mid \langle NUM \rangle \mid (\langle Expr \rangle)$$

```
1  public IExpr parseFactor() {
2    // <Factor> ::= <VAR>
3    String lexeme = getLexeme(REGEXP_VAR);
4    if (lexeme) { return var(lexeme); }
5    // <Factor> ::= <NUM>
6    lexeme = getLexeme(REGEXP_NUM);
7    if (lexeme) { return cnst(Integer.parseInt(lexeme)); }
8    // <Factor> ::= ( <Expr> )
9    lexeme = getLexeme(REGEXP_OPEN_PAREN);
10   if (lexeme) {
11     IExpr e = parseExpr();
12     if (e != null) {
13       lexeme = getLexeme(REGEXP_CLOSE_PAREN);
14       if (lexeme != null) {
15         return e;
16   } } }
17   throw new IllegalArgumentException("Cannot parse input");
18 }
```

Beispiel (Problem): Methode für $\langle Expr \rangle$

- Verfahren funktioniert gut für $\langle Factor \rangle$, weil jede Alternative mit einem anderen Lexem beginnt.
- Bei den anderen Variablen ist das nicht der Fall.
- Betrachte das Beispiel $\langle Expr \rangle$

$$\langle Expr \rangle ::= \langle Expr \rangle + \langle Term \rangle \mid \langle Term \rangle$$

```
1 public IExpr parseExpr() {
2   // <Expr> ::= <Expr> + <Term>
3   // not clear how to check for the other production
4   // unfortunately, this production is left-recursive:
5   IExpr left = parseExpr();
6   // oops, this recursive call does not terminate!
7   // ...
8 }
```

- Um dieses Problem zu vermeiden, muss die BNF ein letztes Mal umstrukturiert werden.

Elimination von Linksrekursion

Die Produktionen für $\langle Expr \rangle$ und $\langle Term \rangle$ sind **linksrekursiv**, d.h., die gleiche Variable taucht direkt am Anfang der rechten Seite einer Regel auf.

$$\begin{array}{lcl} \langle Expr \rangle & ::= & \langle Expr \rangle + \langle Term \rangle \mid \langle Term \rangle \\ \langle Term \rangle & ::= & \langle Term \rangle * \langle Factor \rangle \mid \langle Factor \rangle \end{array}$$

Diese Produktionen können umgeformt werden, so dass sie nicht mehr linksrekursiv sind, aber dass die gleiche Sprache erkannt wird. Dabei steht ε für eine leere rechte Regelseite.

$$\begin{array}{lcl} \langle Expr \rangle & ::= & \langle Term \rangle \ \langle Expr1 \rangle \\ \langle Expr1 \rangle & ::= & \varepsilon \mid + \langle Term \rangle \langle Expr1 \rangle \\ \langle Term \rangle & ::= & \langle Factor \rangle \ \langle Term1 \rangle \\ \langle Term1 \rangle & ::= & \varepsilon \mid * \langle Factor \rangle \langle Term1 \rangle \end{array}$$

Unterscheiden der Alternativen durch Lookahead

$$
\begin{array}{lcl}
\langle Expr \rangle & ::= & \langle Term \rangle \ \langle Expr1 \rangle \\
\langle Expr1 \rangle & ::= & \varepsilon \mid + \ \langle Term \rangle \langle Expr1 \rangle \\
\langle Term \rangle & ::= & \langle Factor \rangle \ \langle Term1 \rangle \\
\langle Term1 \rangle & ::= & \varepsilon \mid * \ \langle Factor \rangle \langle Term1 \rangle \\
\langle Factor \rangle & ::= & \langle VAR \rangle \mid \langle NUM \rangle \mid (\ \langle Expr \rangle \ )
\end{array}
$$

Erkennen der Regel-Alternativen durch Lookahead

- $\langle Factor \rangle$: ok
- $\langle Expr \rangle$, $\langle Term \rangle$: nur eine Alternative
- $\langle Expr1 \rangle$: durch Testen des folgenden Lexems (Lookahead)
  - falls + in der Eingabe, dann zweite Regel
  - falls $*$ oder) in der Eingabe, dann erste (leere) Regel
- $\langle Term1 \rangle$: durch Testen des folgenden Lexems (Lookahead)
  - falls $*$ in der Eingabe, dann zweite Regel
  - falls) in der Eingabe, dann erste (leere) Regel

Beispiel: Methode für $\langle Term1 \rangle$

```
1  public IExpr parseTerm1(IExpr left) {
2    String lexeme = getLexeme (REGEXP_STAR);
3    if (lexeme != null) {
4      IExpr right = parseFactor();
5      if (right != null) {
6        return parseTerm1 (product (left, right));
7      }
8      throw new IllegalArgumentException ("cannot parse Term1");
9    }
10   // should test lookahead for nice error message
11   return left;
12 }
```

- parseTerm1 erhält beim Aufruf den linken Faktor als Argument
- Dadurch wird $*$ links-assoziativ
- parseExpr1 wird analog implementiert.

Alles zusammen...

```
1  public class Parser {
2    private IScanner scan;
3    public Parser(IScanner scan) { this.scan = scan; }
4
5    // delegate to scanner
6    private String getLexeme(String re) { return scan.getLexeme(re); }
7    private boolean lookingAt(String re) { return scan.lookingAt(re); }
8
9    public IExpr parseExpr() {
10     IExpr left = parseTerm();
11     if (left != null) { return parseExpr1(left); }
12     throw new IllegalArgumentException ("cannot parse expression");
13   }
14   // further parseN methods ...
15 }
```