

Compilerbau

Martin Plümicke

WS 2021/22

Agenda

Überblick Vorlesung

- Literatur

Compiler Überblick

Compiler (I)

- Scanner

- Parser

Überblick Funktionale Programmierung

- Einleitung

- Haskell-Grundlagen

- Scanner in Funktionalen Sprachen

- Parser in Funktionalen Sprachen

Compiler (II)

- Abstrakte Syntax

- Abstrakte Syntax Java

- Semantische Analyse/Typecheck

- Antlr

- Codegenerierung

Literatur

 Bauer and Höllerer.
Übersetzung objektorientierter Programmiersprachen.
Springer-Verlag, 1998, (in german).

 Alfred V. Aho, Ravi Lam, Monica S.and Sethi, and Jeffrey D. Ullman.
Compiler: Prinzipien, Techniken und Werkzeuge.
Pearson Studium Informatik. Pearson Education Deutschland, 2.
edition, 2008.
(in german).

 Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman.
Compilers Principles, Techniques and Tools.
Addison Wesley, 1986.

 Reinhard Wilhelm and Dieter Maurer.
Übersetzerbau.
Springer-Verlag, 2. edition, 1992.
(in german).

Literatur II



James Gosling, Bill Joy, Guy Steele, Gilad Bracha, and Alex Buckley.
The Java[®] Language Specification.

The Java series. Addison-Wesley, Java SE 8 edition, 2014.



Tim Lindholm, Frank Yellin, Gilad Bracha, and Alex Buckley.
The Java[®] Virtual Machine Specification.

The Java series. Addison-Wesley, Java SE 8 edition, 2014.



Bryan O'Sullivan, Donald Bruce Stewart, and John Goerzen.
Real World Haskell.

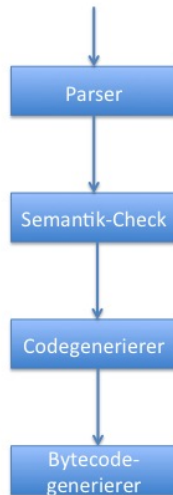
O'Reilly, 2009.



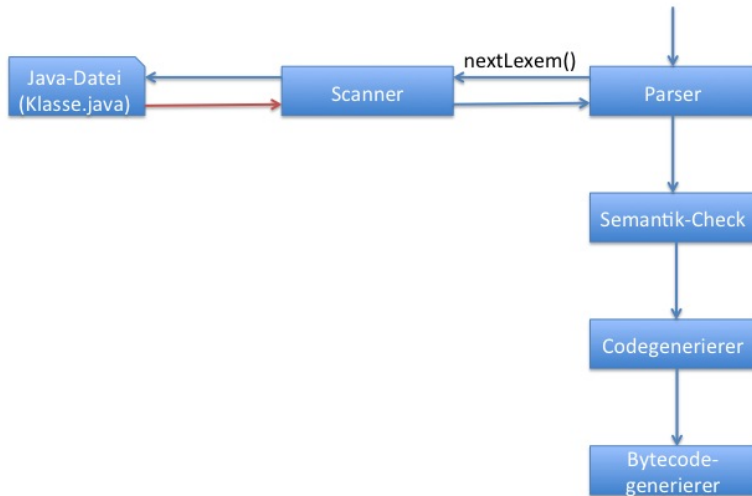
Peter Thiemann.
Grundlagen der funktionalen Programmierung.

Teubner, 1994.

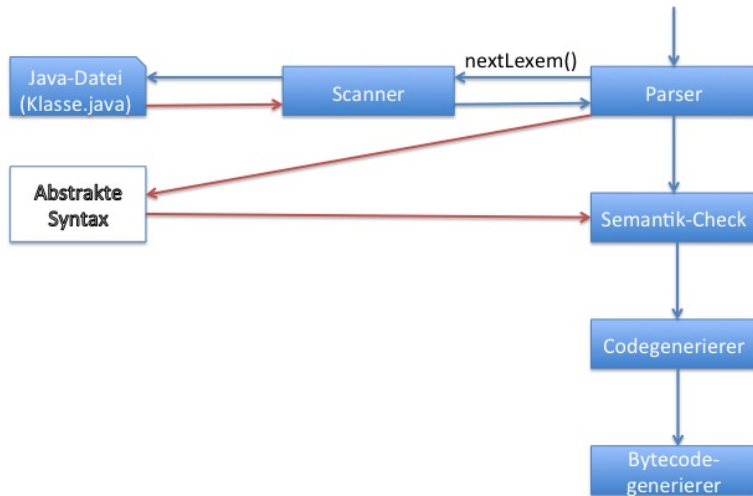
Compiler Überblick



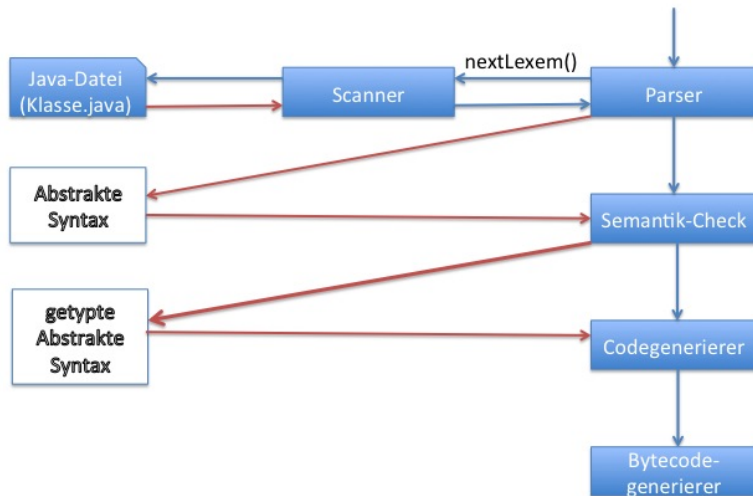
Compiler Überblick



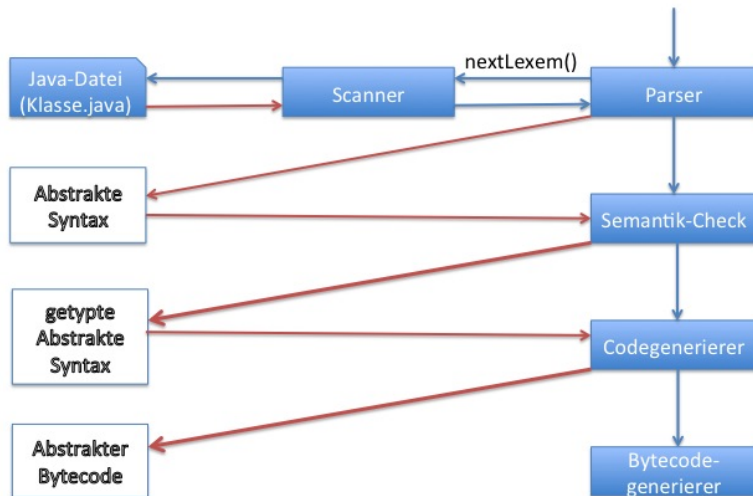
Compiler Überblick



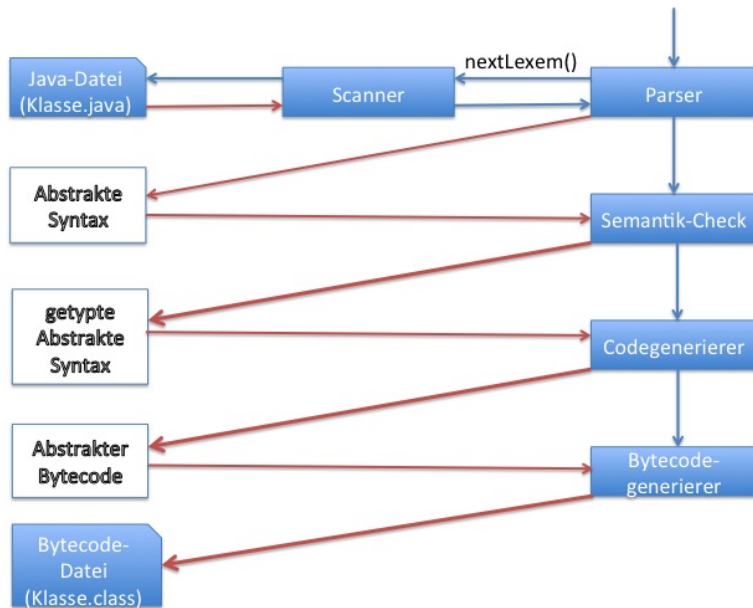
Compiler Überblick



Compiler Überblick



Compiler Überblick



Compilerbau

Martin Plümicke

WS 2021/22

Parser Generator

Organisatorisches

- ▶ Ablauf der Vorlesung gliedert sich in 3 Teile:
 1. Vorlesung
 2. Übungsblatt (in 2er Gruppen bearbeiten)
 3. Vorstellen/Besprechung der Ergebnisse
- ▶ Bearbeiten der Aufgaben mit eigenem Laptop (Jede Zweiergruppe sollte einen Rechner besitzen)

Literatur

- ▶ Offizielle ANTLR-Website unter: <https://www.antlr.org/>
- ▶ *The Definitive ANTLR 4 Reference* von Terence Parr, ISBN: 978-1-93435-699-9
- ▶ Readme auf Github: <https://github.com/antlr/antlr4/blob/master/doc/index.md>

Entwicklungsumgebung

ANTLR-Jar Datei

<https://www.antlr.org/download/antlr-4.9.2-complete.jar>
lässt sich zusammen mit jeder Entwicklungsumgebung auf jedem gängigen Betriebssystem nutzen.

Plugin ANTLR lässt sich per Plugin in gängige Java IDEs integrieren:

IntelliJ <http://plugins.jetbrains.com/plugin/7358?pr=idea>

Eclipse <https://github.com/jknack/antlr4ide>

Vorzüge von ANTLR

- ▶ Trennung von Grammatik und Verarbeitungslogik
 - ▶ Programm verarbeitet den geparsen Syntaxbaum. Kein Java-Source Code innerhalb der Grammatik
 - ▶ Bessere Integration des SourceCodes in die IDE
 - ▶ Leichter anpassbar/erweiterbar
 - ▶ Arbeiten mit bekannter Java Syntax
- ▶ Gutes Tooling: Plugins für gängige Java IDEs, wie Eclipse und IntelliJ IDEA
 - ▶ Syntax Highlighting
 - ▶ Gute Fehlerausgabe

G4-Grammatik: Syntax

- ▶ Comments sind gleich wie in Java (`/**` `/**`)
- ▶ Tokens/Terminals beginnen mit Großbuchstaben, Non-Terminals mit Kleinbuchstaben
- ▶ Literale in Single-Quotes (`'literal'`)
- ▶ Regeln bestehen aus Namen für Token, Literale und den Zeichen (`|` `.` `*` `+` `?`)
 - ▶ `|` entspricht *Oder*
 - ▶ `.` `*` `+` `?` verhalten sich wie bei Regulären Ausdrücken
 - ▶ `~[A]` entspricht allen 16-bit wertigen Zeichen außer *A*

Syntax Veranschaulichung

```
//Datei GrammatikBeispiel.g4
//Name der Grammatik (gleich wie Dateiname):
grammar GrammatikBeispiel;
regel1 : Token1 Token2;
Token1 : 'Hallo';
Token2 : [a-z]+;
/*
Grammatik parst Texte der Form:

Hallo nameinkleinbuchstaben
*/
Linefeed: '\r'? '\n';
WS : [ \t]+ -> skip; //Leerzeichen ignorieren
```

G4-Grammatik Beispiel

```
grammar Beispiel2;  
s : s '(' s ')' s  
  | TEXT?;  
TEXT : ~[()]+;
```

- (Gibt es ein äquivalent zu dieser Grammatik als regulären Ausdruck?)

Hinweise

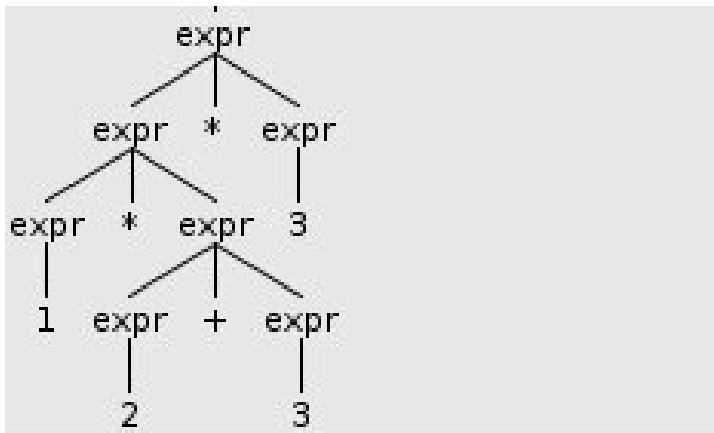
- ▶ Keine separate Datei für Lexeme notwendig. Alle Regeln und Lexeme werden in der G4-Grammatik beschrieben.
- ▶ Der Lexer nimmt immer das erste in der Grammatik auftauchende Lexem. Allgemeinere Terminale also ans Ende der Grammatik verlegen.

```
//Warum kann diese Grammatik "int 1" nicht parsen?  
grammar Number;  
number : integer | float;  
integer : 'int ' Integer;  
float : 'float ' Float;  
  
Float : [0-9.]+;  
Integer : [0-9]+;
```

- ▶ Bei nicht eindeutigen Regeln wird die zuerst spezifizierte Regel bevorzugt
- ▶ ANTLR versucht immer das längste Lexem auszuwählen

Falsch:

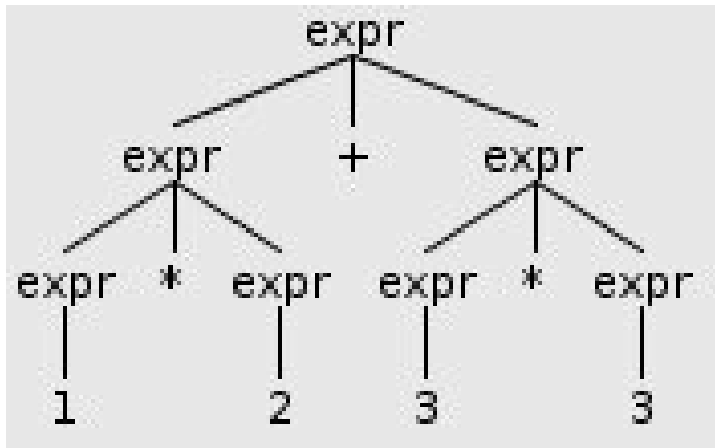
```
expr : expr '+' expr | expr '*' expr | Number;
```



Baum beim Parsen von $1 * 2 + 2 * 3$

Korrekt:

```
expr : expr '*' expr | expr '+' expr | Number;
```



Baum beim Parsen von $1 * 2 + 2 * 3$

Übungsblock 1

Übungsblatt 1: Aufgabe 1 und 2

- ▶ Link zum Übungsblatt:
`http://www2.ba-horb.de/~stan/%C3%BCbung1.pdf`
- ▶ IntelliJ-Download: `https://www.jetbrains.com/idea/download`
- ▶ Download der ANTLR-Java-Library
`https://www.antlr.org/download/antlr-4.9.3-complete.jar`

Tip: Installation des ANTLR Plugins in IntelliJ

- ▶ File → Settings → Suche nach *Antlr v4 Grammar Plugin*
Oder: Download ANTLR-Plugin: `https://plugins.jetbrains.com/plugin/download?pr=idea&updateId=26416`

ANTLR-Library dem Projekt hinzufügen

- ▶ antlr-complete.jar ins Projektverzeichnis kopieren
- ▶ File → Project Structure → Libraries
- ▶ anschließend '+' → Java → antlr-complete.jar auswählen

Beispielimplementierung für Aufgabe 3.b

Grammatik für Expressions (Fehlerhaft)

```
grammar IntExpression;  
s : expr;  
  
expr : expr ADD expr | expr MUL expr  
      | expr SUB expr | '(' expr ')'  
      | Number;  
  
MUL : '*' ;  
ADD : '+' ;  
SUB : '-' ;  
  
Number : [0-9]+;  
WS : [ \t\r\n] -> skip;
```

Methode zum Einlesen und Parsen eines Strings aus System.in

```
public static void main(String[] args) throws Exception {
    CharStream input = CharStreams.fromString("100+2*3");
    IntExpressionLexer lexer = new IntExpressionLexer(input);
    CommonTokenStream tokens = new CommonTokenStream(lexer);
    IntExpressionParser parser = new IntExpressionParser(tokens);
    IntExpressionParser.StartContext tree = parser.start(); //Parsen

    ExpressionCalculator calc = new ExpressionCalculator();
    int ergebnis = calc.calculate(tree.expr()); // initiate walk of tree with
        listener
    System.out.println(ergebnis);
}
```

ExprAdapter:

```
class ExpressionCalculator{
    int calculate(IntExpressionParser.ExprContext ctx){
        if (ctx.MUL()!=null){
            return this.calculate(ctx.expr(0)) * this.calculate(ctx.expr(1));
        }else
        if (ctx.ADD()!=null){
            return this.calculate(ctx.expr(0)) + this.calculate(ctx.expr(1));
        }else
        if (ctx.Number()!=null){
            return Integer.parseInt(ctx.Number().toString());
        }
        return 0;
    }
}
```

Einlesen von Texten

- ▶ **CharStreams** Doku: <http://www.antlr.org/api/Java/org/antlr/v4/runtime/CharStreams.html>
- ▶ Factory zur Generierung von CharStream's
- ▶ Kann String, InputStream, File und viele weitere Eingabeformate in das von ANTLR benutzte CharStream konvertieren

```
//Beispiel:  
CharStream input = CharStreams.fromFileName("/pfad/zu/Datei");
```

Übungsblock 2

Übungsblatt 1: Aufgabe 3

Aufbau des ParseTrees

- ▶ ANTLR generiert zu jeder Regel in der Grammatik eine eigene Klasse
- ▶ Der Aufbau der Klasse wird direkt von dieser Regel abgeleitet
- ▶ Der ParseTree baut sich aus diesen Klasse auf

Beispiel 1:

```
regel : unterRegel1 | unterRegel2
```

wird durch die Klasse `RegelContext` repräsentiert, welche folgende Methoden enthält:

```
UnterRegel1Context unterRegel1();  
UnterRegel2Context unterRegel2();
```

Aufbau des ParseTrees

Beispiel 2:

```
regel : unterRegel+
```

wird durch die Klasse `RegelContext` repräsentiert, welche folgende Methoden enthält:

```
List<UnterRegelContext> unterRegel();
```

Parse Tree Walker

- ▶ Der generierte Parser liest den ihm übergebenen Text zu einem Syntaxbaum ein
- ▶ Dieser kann anschließend mittels der von ANTLR angebotenen API durchlaufen werden.
- ▶ Der Syntaxbaum wird dabei mittels einer Tiefensuche durchlaufen
- ▶ Zu jeder Regel in der Grammatik kann eine Methode spezifiziert werden, welche beim durchlaufen des zugehörigen Knotens im Syntaxbaum aufgerufen wird

Implementierung

- ▶ Der Parser generiert einen Parse Tree aus dem eingelesenen Text
- ▶ Dieser kann anschließend mittels eines ParseTreeListeners durchlaufen werden
- ▶ Zur Implementierung eines ParseTreeListener sollte die Klasse BaseListener erweitert werden
 - ▶ Diese Klasse verfügt über alle Methoden, welche beim Durchlaufen des Syntaxbaums aufgerufen werden.
 - ▶ Um gewisse Knoten zu verarbeiten müssen diese Methoden überschrieben werden:

```
@Override  
public void enter<NameDerRegel>(DocumentContext ctx) {  
    //...  
}
```

Implementierung

- ▶ Eine Instanz dieses Listeners kann anschließend einem ParseTreeWalker übergeben werden. Dieser durchläuft den Syntaxbaum und ruft die entsprechenden Methoden im Listener auf.
- ▶ Beispiel für das Aufrufen eines ParseTreeWalkers für eine Grammatik namens 'Beispiel':

```
// Text von Standard-Input einlesen :
ANTLRInputStream input = new ANTLRInputStream(System.in);
// Text parsen :
BeispielLexer lexer = new ArrayInitLexer(input);
CommonTokenStream tokens = new CommonTokenStream(lexer);
BeispielParser parser = new ArrayInitParser(tokens);
ParseTree tree = parser.init(); // Das Parsen mit der Regel init beginnen
// ParseTreeWalker generieren, welcher die Methoden im Listener aufruft:
ParseTreeWalker walker = new ParseTreeWalker();
// Listener erstellen:
SelbstgeschriebenerListener listener = new SelbstgeschriebenerListener();
// Den Syntaxbaum durchlaufen:
walker.walk(listener, tree);
```

Linksrekursive Grammatik

- ▶

```
expr : expr op expr  
expr : Number  
  
Number : [0-9]+  
op : '+' | '-' | ...
```
- ▶ Die linksrekursion kann mit dem *-Operator beseitigt werden:

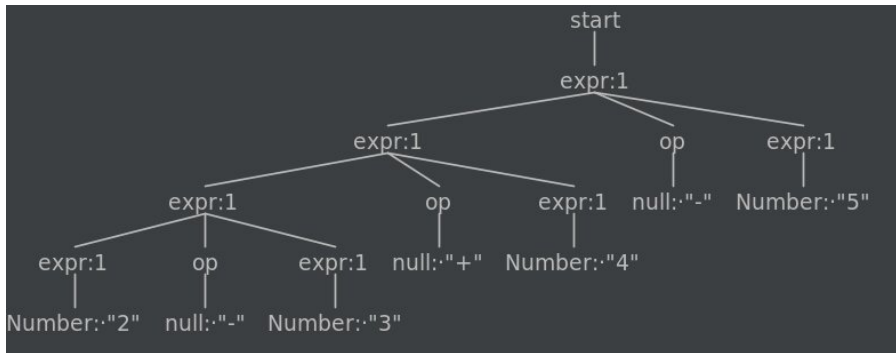
- ▶

```
expr : Number (op Number)*  
...
```

- ▶ ergibt dann aber einen anderen ParseTree

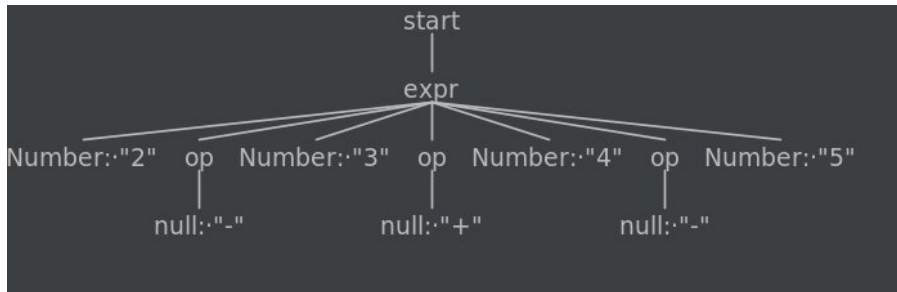
Linksrekursive Grammatik

- ▶ Grammatik: $\text{expr} : \text{expr op expr}$
- ▶ Eingabe: 2-3+4-5
- ▶ ParseTree:



Linksrekursive Grammatik

- ▶ Grammatik: $\text{expr} : \text{Number} (\text{op} \text{Number})^*$
- ▶ Eingabe: $2-3+4-5$
- ▶ ParseTree:



Linksrekursive Grammatik

- ▶ ANTLR ist in der Lage linksrekursive Grammatiken automatisch aufzulösen
- ▶ Dies funktioniert unter Umständen nicht immer:

Markdown

Informationen zur Markdown Sprache:

Markdown-Tutorial

<https://guides.github.com/features/mastering-markdown/>

Markdown-Syntax

<http://daringfireball.net/projects/markdown/syntax>

Markdown-Editor <https://stackedit.io/>

Übungsblock 3

- ▶ Link zum Übungsblatt:

<http://www2.ba-horb.de/~stan/%C3%BCbung2.pdf>