
DHBW
Stuttgart

Organisatorisches

- Ablauf der Volesung 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 mindestens 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-4.12.0-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`

Parser Generator

Vorzüge von ANTLR

- Trennung von Grammatik und Verarbeitungslogik
  - Programm verarbeitet den geparsten 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/Terminale beginnen mit Großbuchstaben, Non-Terminale 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.

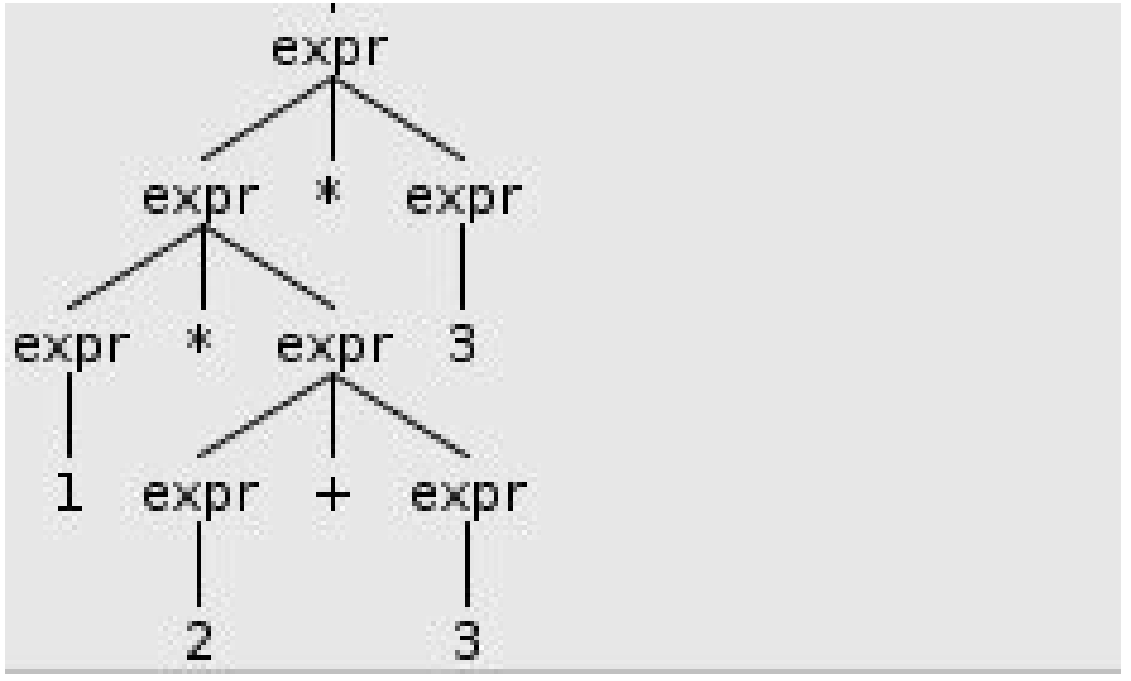
```
//Warum kann diese Grammatik "int 1" nicht parsen?
grammar Number;
number : integer | float;
integer : 'int␣' Integer;
float : 'float␣' Float;

Float : [0-9\.];
Integer : [0-9];
```

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

Falsch:

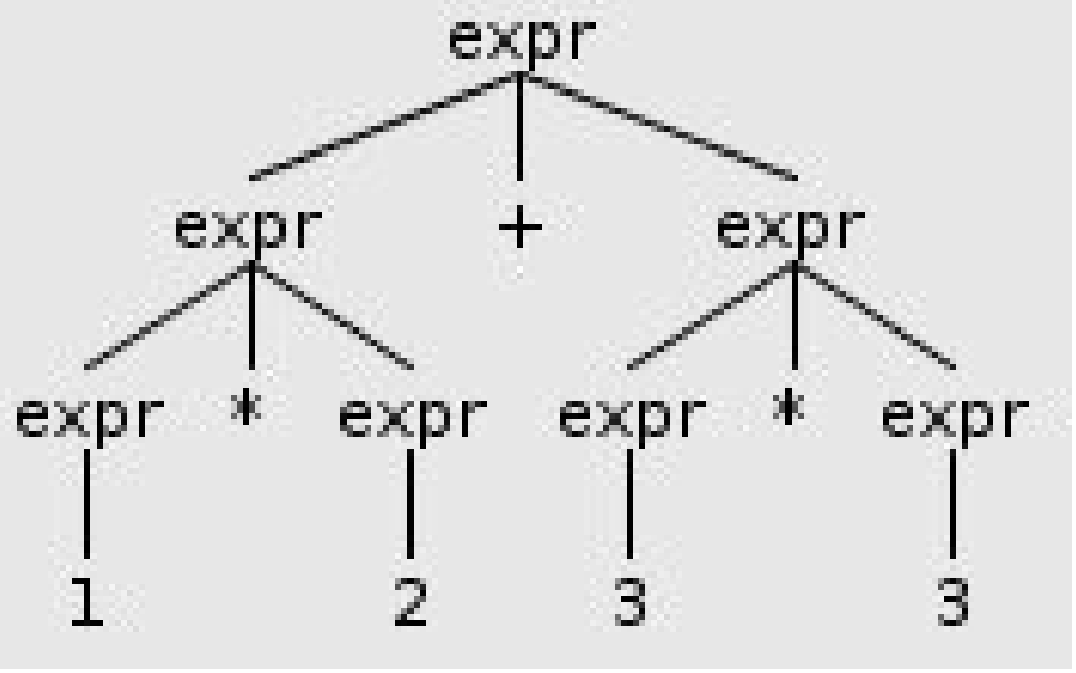
```
expr : expr '+' expr | expr '*' expr | Number;
```



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-4.12.0-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 unterRegel1();
```

Aufbau des ParseTrees

Beispiel 2:

```
regel : unterRegel+
```

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

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

Decaf Programmiersprache

- Simple Programmiersprache
- Java-like Syntax
- aber keine Klassen und Vererbung

Beispiel:

```
def int add(int x, int y)
{
    return x + y;
}

def int main()
{
    int a;
    a = 3;
    return add(a, 2);
}
```

Decaf: Syntax (1)

$$
\begin{aligned}
\mathit{Program} &\rightarrow (\mathit{Var} \mid \mathit{Method})* \\
\\
\mathit{Var} &\rightarrow \mathit{Type}\ \texttt{ID}\ \texttt{';'} \\
\mathit{Method} &\rightarrow \texttt{'def'}\ \mathit{Type}\ \texttt{ID}\ \texttt{'('}\ \mathit{Params}?\ \texttt{')'}\ \mathit{Block} \\
\mathit{Type} &\rightarrow \texttt{'int'} \mid \texttt{'bool'} \mid \texttt{'void'} \\
\\
\mathit{Block} &\rightarrow \texttt{'\{'}\ \mathit{Var}*\ \mathit{Stmt}*\ \texttt{'\}'} \\
\mathit{Stmt} &\rightarrow \mathit{Loc}\ \texttt{'='}\ \mathit{Expr}\ \texttt{';'} \\
&\quad \mid \mathit{MethodCall}\ \texttt{';'} \\
&\quad \mid \texttt{'if'}\ \texttt{'('}\ \mathit{Expr}\ \texttt{')'}\ \mathit{Block}\ (\texttt{'else'}\ \mathit{Block})? \\
&\quad \mid \texttt{'while'}\ \texttt{'('}\ \mathit{Expr}\ \texttt{')'}\ \mathit{Block} \\
&\quad \mid \texttt{'return'}\ \mathit{Expr}?\ \texttt{';'}
\end{aligned}
$$

Decaf: Syntax (2)

$$
\begin{aligned}
\mathit{Expr} &\rightarrow \mathit{Expr}\ \mathit{Operator}\ \mathit{Expr} \\
&\mid\ \texttt{'('}\ \mathit{Expr}\ \texttt{')'} \\
&\mid\ \mathit{Loc} \\
&\mid\ \mathit{MethodCall} \\
&\mid\ \mathit{Constant} \\
\mathit{Loc} &\rightarrow \texttt{ID} \\
\\
\mathit{MethodCall} &\rightarrow \texttt{ID}\ \texttt{'('}\ \mathit{Args}?\ \texttt{')'} \\
\mathit{Args} &\rightarrow \mathit{Expr}\ (\texttt{','}\ \mathit{Expr})* \\
\\
\mathit{Constant} &\rightarrow \texttt{NUMBER} \mid \texttt{'true'} \mid \texttt{'false'} \\
\mathit{Operator} &\rightarrow \texttt{'+'} \mid \texttt{'-'} \mid \texttt{'*'}
\end{aligned}
$$

Decaf: Abstrakte Syntax (1)

```
Program                     (enthält Variablen und Funktionen)
Variable
Function                    (enthält einen Block)
Block                       (enthält Variablen und Statements)
Statement
    Assignment              (Location = Expression)
    VoidFunctionCall        (wrapper für FunctionCall)
    IfElse                  (eine Expression und zwei Blöcke)
    WhileLoop               (enthält Expression und Block)
    Return                  (enthält Expression)
    VoidReturn              (return ohne Expression)
```

Decaf: Abstrakte Syntax (2)

```
Expression
    BinaryExpr                  (enthält zwei Expressions)
    Location                    (Variablenzugriff)
    FunctionCall                (enthält mehrere Expressions)
    Literal
        StringLiteral
        IntLiteral
        BoolLiteral
```

AST Implementierung (1)

- Implementierng als `record` Typen
  - Immutable (unveränderliche) Datenstruktur
  - Erstellt `get` und `set` Methoden automatisch

```
record Program(List<Variable> variablen, List<Function> funktionen) {}
record Variable(String name, Type type) {}
record Function(Type type, String name, List<Variable> params, Block block)

interface Expression {}
record BinaryExpr(Expression left, Operator op, Expression right) implements Expression {}
enum Operator { ADD, SUB, MUL }
record Location(String varName) implements Expression {}
record FunctionCall(String varName, List<Expression> params) implements Expression {}
record StringLiteral(String value) implements Expression {}
record IntLiteral(Integer value) implements Expression {}
record BoolLiteral(Boolean value) implements Expression {}
```

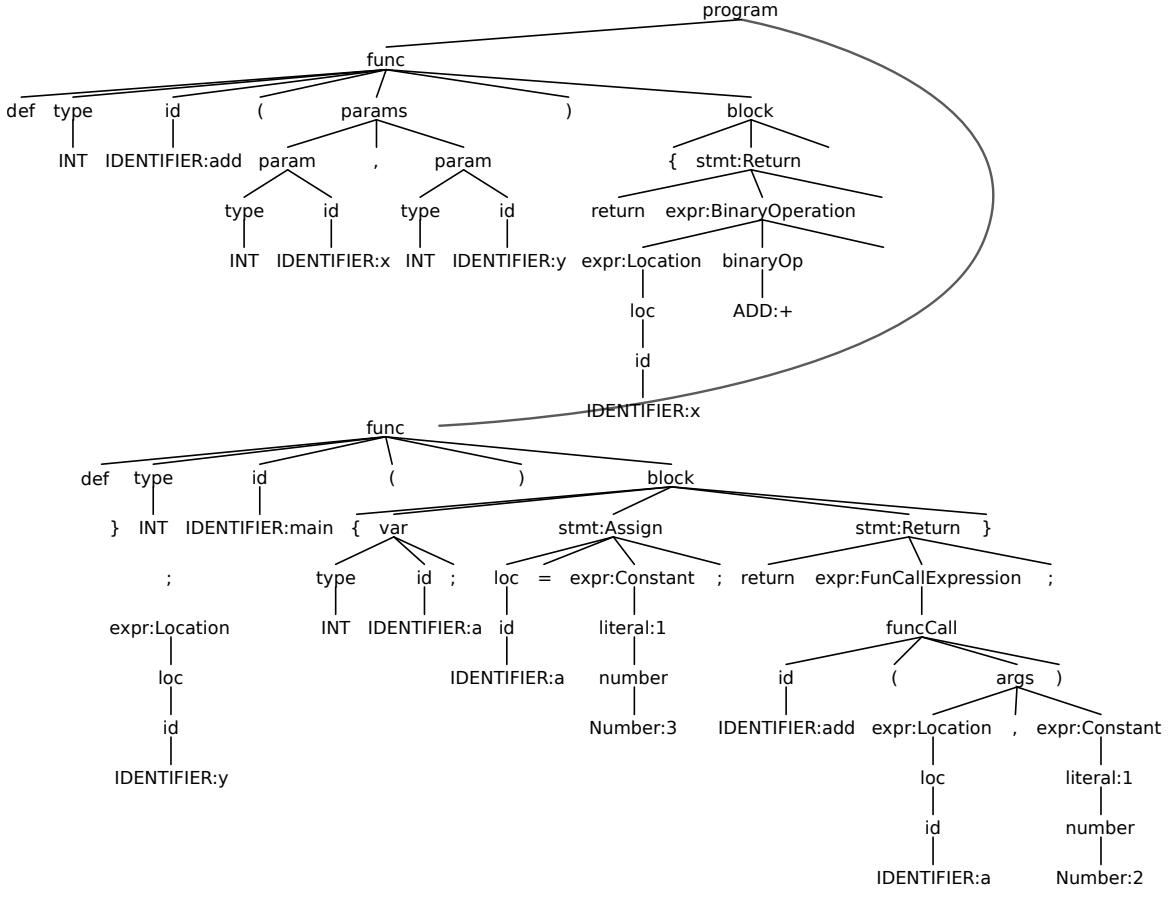
AST Implementierung (2)

- Statements:

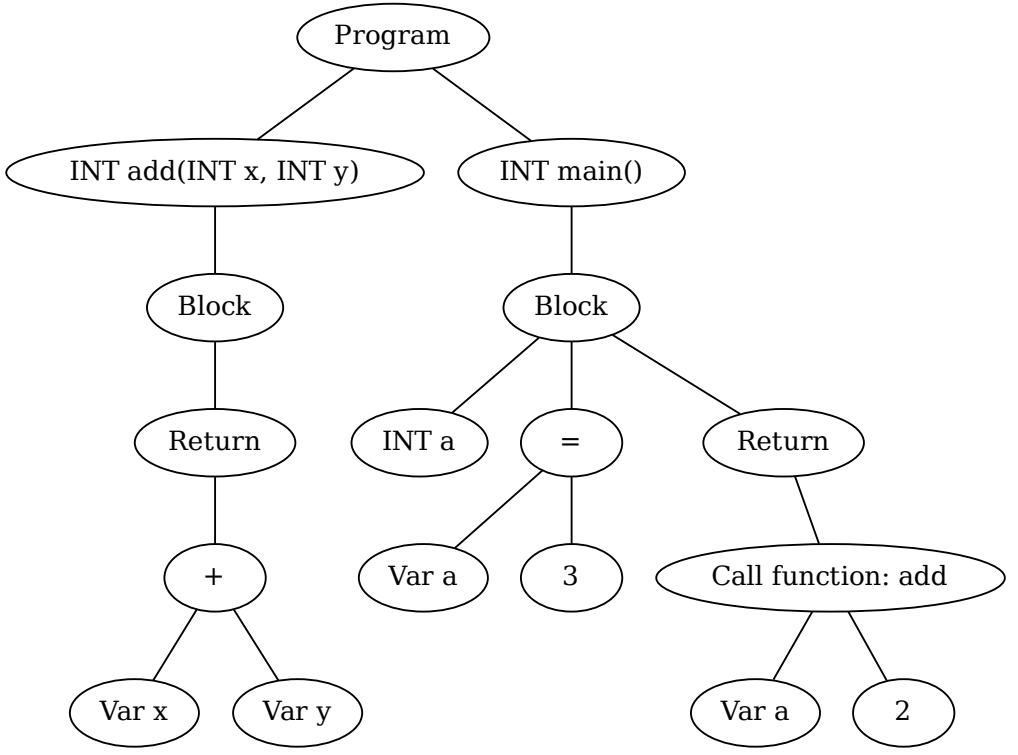
```
interface Statement {}
record Assignment(Location loc, Expression value) implements Statement {}
record VoidFunctionCall(FunctionCall expr) implements Statement {}
record ReturnVoid() implements Statement {}
record Return(Expression expr) implements Statement {}
record IfElse(Expression cond, Block ifBlock, Block elseBlock) implements Statement{}
record While(Expression cond, Block block) implements Statement {}
```

Umwandlung: ParseTree → AST

Parse Tree



Abstract Syntax Tree



ASTVisitor

- ANTLR erzeugt eine `Vistor` und eine `BaseVisitor` Klasse.
- Beispiel:

```
grammar Beispiel;

regel : unterRegel1 | unterRegel2;
unterRegel1 : "Hallo";
unterRegel2 : "Besucher";
```

generiert die `BeispielBaseVisitor` Klasse:

```
interface TestVisitor<T> {
    <T> visitRegel(RegelContext ctx);
    <T> visitUnterRegel1(UnterRegel1Context ctx);
    <T> visitUnterRegel2(UnterRegel2Context ctx);
}
```

ASTVisitor

Problem: `StmtContext` im `ParseTree` kann verschiedene Statements darstellen.

Kann beispielsweise durch Typcasts gelöst werden:

```
//Methode soll aus dem ParseTree Element Statement ein AST Element generieren:
Statement generate(StatementContext ctx){
    if(ctx instanceof WhileContext){
        WhileContext wCtx = (WhileContext) ctx;
        return ... //Generate While
    }
    if(ctx instanceof ReturnContext){
    ...
}
```

ASTVisitor

Problem: `StmtContext` im `ParseTree` kann verschiedene Statements darstellen.

Lösung: Mithilfe der `BaseVisitor` Klasse geht es auch ohne Casts:

```
class StatementGenerator extends DecafBaseVisitor<Statement> {
    Statement visitWhile(WhileContext ctx){
        ...
    }
    Statement visitReturn(ReturnContext ctx){
            ...
    }
}
```

ASTVisitor - Grammatik Labels

```
stmt : loc '=' expr ';'                                #Assign
     | funcCall ';'                                    #FunctionCall
     | 'if' '(' expr ')' block ('else' block)?         #If
     | 'while' '(' expr ')' block                      #While
     | 'return' expr ';'                               #Return
     | 'return' ';'                                    #ReturnVoid
     | 'break' ';'                                     #Break
     | 'continue' ';'                                  #Continue
     ;
```

- Zusätzliche Strukturierung durch `#Labels`
- ANTLR generiert einen zusätzlichen Visitor für jedes Label

Übungsblock 3

Übungsblatt 2: Aufgabe 1 und 2

- Link zum Übungsblatt: `http://www2.ba-horb.de/~stan/%C3%BCbung2.pdf`
- Link zum Maven-Download: `https://maven.apache.org/download.cgi`