# Compilerbau Projekt

Till Köpff, Kim Schuster und Matthias Wallner
Eberhard Karls Universität Tübingen

Wintersemester 2022/2023

## 1 Spezifikation

**Deklaration:**

- $\Sigma$: Eingabe-Alphabet
- JC: Menge aller syntaktisch korrekten Java-Klassen der Untermenge von Java
- BC: Menge aller Bytecode-Files

**Eingabe:** $p \in \Sigma^*$
**Vorbedinung:** $\emptyset$
**Ausgabe:** $bc \in \text{BC}^* \cup \{error\}$
**Nachbedinungen:** Falls $p \in (\text{JC})^*$, so ist $bc \in (\text{BC})^*$ und p wird nach $bc$ übersetzt wie es durch die Sprache Java definiert ist. Falls $p \notin (\text{JC})^*$, so ist $bc = error$.

**Unsere Projektlösung ist unter https://github.com/tkpf/MiniJavaCompiler [1] auffindbar!**

## 2 Abstrakter Syntaxbaum

Der Abstrakte Syntaxbaum ist nach Vorbild der in der Vorlesung vorgestellten abstrakten Syntax konzipiert worden. Die Syntax aus den Vorlesungsfolien ist in Bild 1 zu erkennen. Vererbung wird nicht unterstützt. Demnach ist die Expression *Super* nicht implementiert, da der Verweis auf die Objekt-Klasse keinen Mehrwert für den Compiler bringt.

```
data Class = Class(Type, [FieldDecl], [MethodDecl])

data FieldDecl = Field(Type, String)

data MethodDecl = Method(Type, String,[(Type,String)], Stmt)

data Stmt = Block([Stmt])
          | Return( Expr )
          | While( Expr , Stmt )
          | LocalVarDecl(Type, String)
          | If(Expr, Stmt , Maybe Stmt)
          | StmtExprStmt(StmtExpr)

data StmtExpr = Assign(String, Expr)
              | New(Type, [Expr])
              | MethodCall(Expr, String, [Expr])
```

```
data Expr = This
          | Super
          | LocalOrFieldVar(String)
          | InstVar(Expr, String)
          | Unary(String, Expr)
          | Binary(String, Expr, Expr)
          | Integer(Integer)
          | Bool(Bool)
          | Char(Char)
          | String(String)
          | Jnull
          | StmtExprExpr(StmtExpr)

type Prg = [Class]
```

Abbildung 1: Ungetypte Abstrakte Syntax aus der VL

Ein *Programm* besteht aus verschiedenen *Klassen*, wobei jede Klasse *Felder* und *Methoden* mit *Parametern* besitzt. Felder, Methoden und Parameter besitzt jeweils einen *Typ*. Bild 2 zeigt das entsprechende UML Diagram.

Methoden bestehen aus Statements. Statements werten Expressions aus und bestehen aus StatementExpressions, welche Operationen auf Expressions ausführen. Die UML Diagramme sind den Bildern 3, 4 und 5 zu entnehmen. Die Methode *toString()* dient dem Debugging und Logging.


Abbildung 2: Head of AST

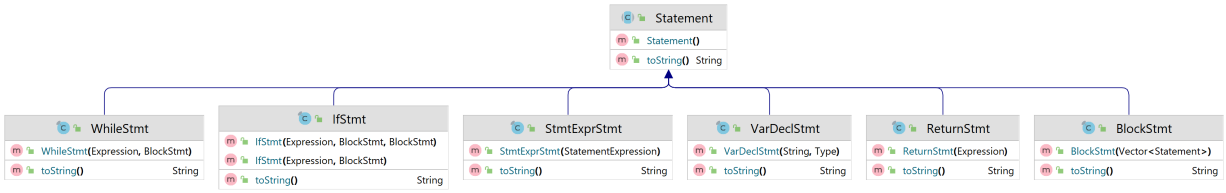


Abbildung 3: Statement Klassen im AST

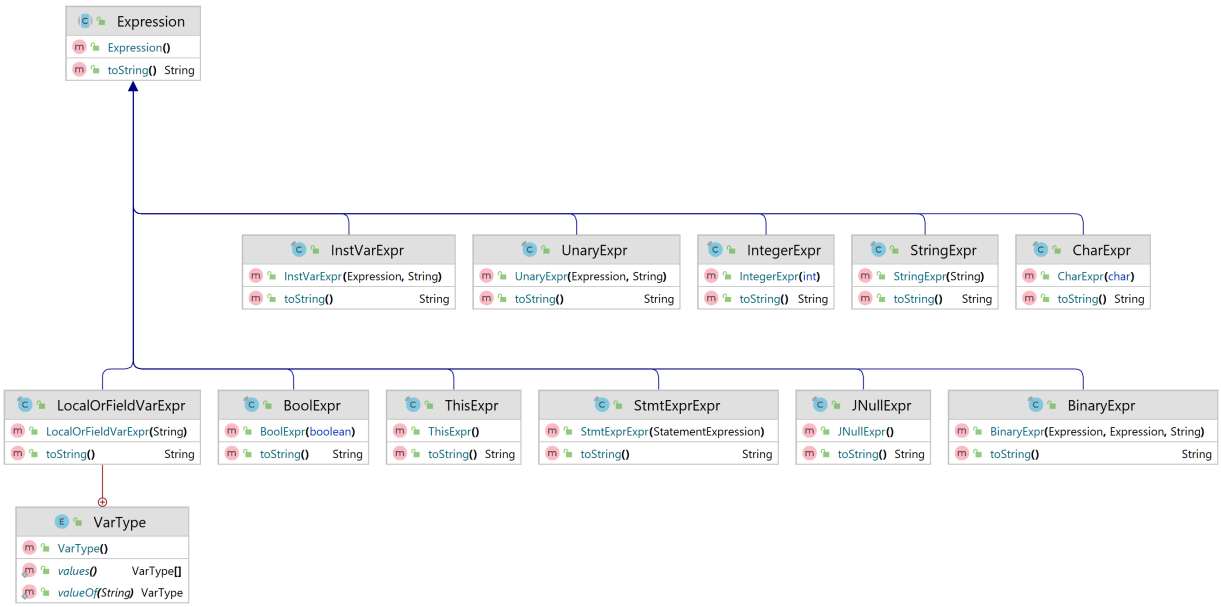


Abbildung 4: Expression Klassen im AST

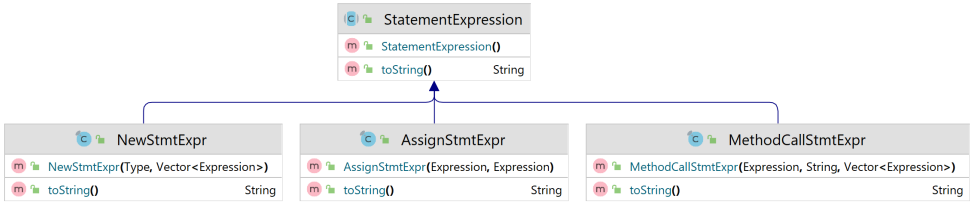


Abbildung 5: StatementExpression Klassen im AST

# 3 Parser (Till Köpff)

Der Parser im Projekt ist via ANTLR [2] Version 4.11.1 und Java-Adapter Klassen realisiert. Die ANTLR Grammatik basiert auf der aktuellen Java 19 ANTLR Grammatik, wurde jedoch erheblich reduziert (MiniJava statt Java) und in entscheidenen Stellen angepasst. Aus den geparsten Grammatik-Kontexten werden im Top-Down Approach die Grammatik-Kontexte in die definierte, abstrakte Syntax übersetzt. Hiefür wird in der aktuellen Kontext-Instanz auf entsprechende Kinderkontexte gecheckt und bei Vorhandensein top-down geparst.
Folgende Eigenschaften des Parsers sind zu erwähnen:

- Konstruktoren werden als Methoden ohne Typ geparst und vom Typechecker erkannt
- Der Parser parst nur das *public* modifier Schlusswort, da Methoden und Felder im MiniJavaCompiler stehts als public angenommen werden
- For-Loops, sowie Arrays werden nicht unterstützt.
- Leere Zeilen mit Semikolon können vom Parser geparst werden (im Klassen-Rumpf als auch im Methoden-Rumpf).

- Aufgrund von ANTLR Eigenheiten bei der Tokenerkennung werden Strings und Chars mit umschließenden Anführungszeichen bzw. Hochkommas geparst. Der TypeLiteralAdapter entpackt die Strings bzw. Chars im Anschluss.
- Die Methode *unescapeJava()* im *TypeLiteralAdapter* sorgt dafür, dass Escape-Sequenzen richtig geparst werden. Andernfalls wird z.B. die newline Zeichenfolge (\n) als einzeln interpretierte Zeichen geparst (\\n entsprechend).
- Identifer können immer mit großem und kleinen Anfangsbuchstaben geparst werden. Gemäß der Best-Practice sollen jedoch Methoden- und Felder-Namen klein und Klassennamen groß geschrieben werden. Identifier können Zahlen enthalten, jedoch nicht als erstes Zeichen (gemäß Java-Spezifikation).
- Die EscapeHatchExpception wird von den Adapter-Klassen implementiert und schützen vor nicht implementierter Verzweigung.

Die Adapter-Klassen, sowie deren Beziehungen zueinander (e.g. verschachtelte Aufrufmöglichkeiten) sind im UML-Diagramm 6 dargestellt. Die Adapterklassen sind statischer Natur.

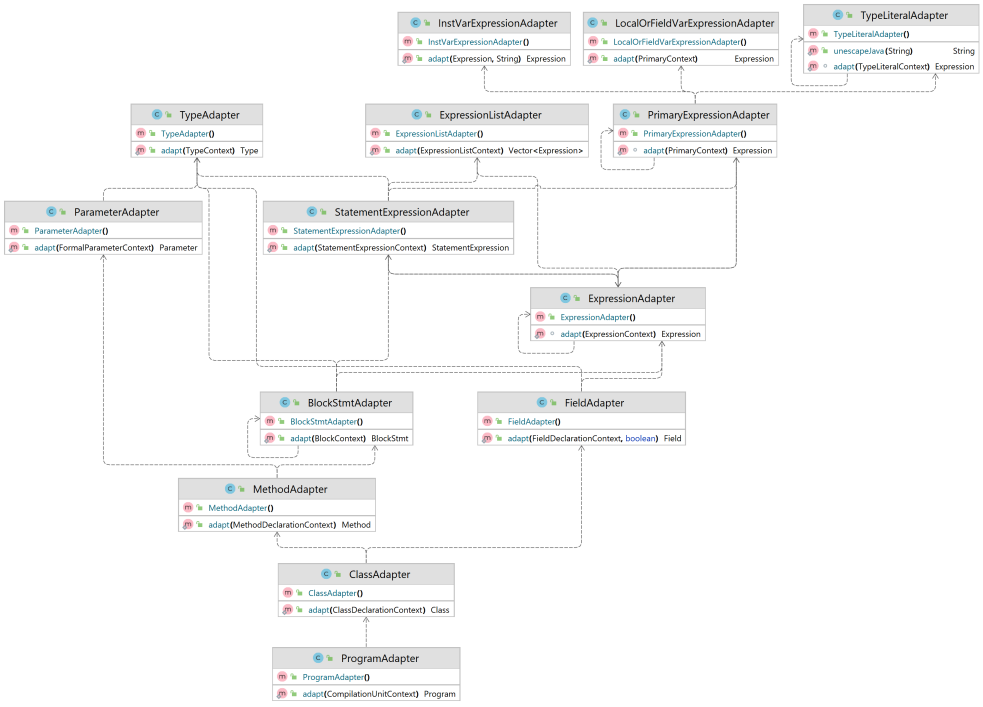


Abbildung 6: UML Diagram der Adapter Klassen

Der ParseTree der Rectangle.java Test-Klasse ist in Anhang A dargestellt.

# 4 Typisierung (Kim Schuster)

Die Typisierung des im vorherigen Schritt erzeugten Syntaxbaums erfolgt unter Zuhilfenahme von *pattern matching* innerhalb von `switch`-Anweisungen. Dabei handelt es sich um ein `Java 19` *preview feature*.

Zunächst erzeugt der Konstruktor der Klasse `GlobalScope` eine globale Umgebung, in der sie die Felder und Methoden aller eingelesenen Klassen sammelt. Methoden werden eindeutig durch ihren Namen und *Signatur* repräsentiert, modelliert durch die Klasse `Signature`. Dadurch wird die Überladung von Methoden unterstützt, da sich gleichnamige Methoden durch die Typen ihrer Parameter unterscheiden lassen. Konstruktoren werden hierbei ebenfalls als Methoden aufgefasst, welche auf den Typ der Klasse abbilden. Hier wird ebenfalls der *Standardkonstruktor* erzeugt, falls noch kein explizit definierter Konstruktor existiert.

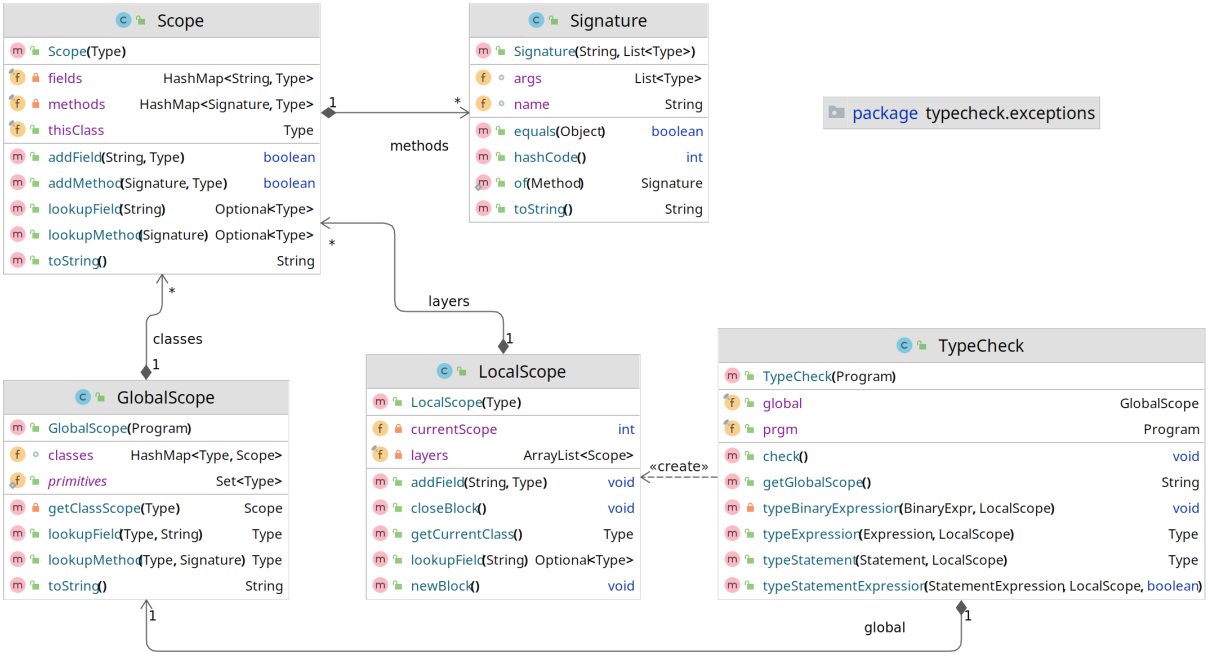


Abbildung 7: UML Diagramm des `typecheck`-Pakets.

Danach wird von Klasse zu Klasse vorgegangen. Zu erst werden *inline*-Zuweisungen aufgelöst, danach Methoden. Pro Methode wird durch die Klasse `LocalScope` eine lokale Umgebung erzeugt und die Parameter der Methode dieser als lokale Felder hinzugefügt.

Die Typisierung erfolgt durch drei Methoden, `typeStatement`, `typeStatementExpression`, und `typeExpression`, jeweils eine pro Oberklasse aller möglichen Syntaxbaumknoten. Diese ermitteln durch *pattern matching* die konkrete Klasse des jeweiligen Knoten und setzten das enthaltene Feld `Type type`. Dabei arbeiten sich die Methoden durch gegenseitige Aufrufe die Äste des Baumes herab, bis ein Blatt direkt typisiert werden kann. Alle beteiligten Klassen sind in Abbildung 7 dargestellt.

## 4.1 Typenableitung

Die Typisierung erfolgt gemäß des *Systems natürlichen Schließens* für die Typenableitung, welches in der begleitenden Vorlesung vorgestellt wurde. Diese Regeln sind im Anhang B abgebildet. Da *Vererbung* durch unser Projekt nicht unterstützt wird beschränkt sich dies vor allem auf das Testen von Typengleichheit, etwa bei Zuweisungen, und bei unären und binären Ausdrücken.

`StatementExpression` Als besonders interessant hervorzuheben ist die Typisierung von `StatementExpressions`. Diese sind nach außen hin immer vom Typ `void`, muessen intern aber genauere Typinformationen enthalten, die für Zuweisungen von Relevanz sind.

Betrachte den folgenden Codeausschnitt:

```
1 {
2     new A();
3     A a = new A();
4     a.fun();
5     int i = a.fun();
6 }
```

Sei `A` eine bereits definierte Javaklasse, die eine Funktion `int fun() { return 1; }` enthält. Bei Zeilen 2–5 handelt es sich jeweils um eine `StatementExpression` und sind damit alle vom Typ `void`. Allerdings benötigen die Zuweisungen in Zeile 3 und 5 intern die Information, dass `new A()` ein Objekt vom Typ `A` und `a.fun()` ein Objekt vom Typ `int` zurückgeben. Bei ersterem ist dies einfach, da der Name „`A`“ auch gleichzeitig der Name ihres Typs ist. Bei Funktionsaufrufen benötigen wir ein zusätzliches Feld `Type innerType`. In diesem kann der Rueckgabetyp der Methode `fun()`, welcher aus der globalen Umgebung ermittelt wird, hinterlegt werden.

Innerhalb des Syntaxbaumes können `StatementExpressions` so nicht einfach Teil von Blöcken oder Zuweisungen sein. In Blöcken müssen diese durch die Klasse `StmtExprStmt` verkapselt werden, diese erhalten ebenfalls den „äußeren“ Typ `void` der `StatementExpression`. Als Teil von Zuweisungen erledigt die Kapselung die Klasse `StmtExprExpr`. Diese muss dafür den „inneren“ Typ, im obigen Beispiel jeweils `A` und `int`, erhalten.

`BlockStmt` Ebenfalls interessant ist die Typisierung von Blockanweisungen. Im folgenden Codebeispiel bilden die Zeilen 1–4 einen Block.

```
1 {
2     int j;
3     j = 1;
4 }
```

Blöcke bestehen aus einer Liste von Anweisungen. Obwohl die Deklaration in Zeile 2 den Typ `int` hat ist der gesamte Block vom Typ `void`. Dies ist auf die Typenableitungsregel für Deklarationen innerhalb von Blöcken zurückzuführen. Dabei wird nämlich der Typ der Deklaration für die Ermittlung des Blocktyps nicht beachtet. Siehe hierzu Anhang B: Block-Statement Regeln - [`Block-LocalVarDecl`].

Ist eine andere Anweisung von einem Typ der nicht `void` ist werden wiederrum andere Anweisungen mit Typ `void` für die Typisierung des Blocks nicht beachtet. Folgener Block ist vom Typ `int`. Dabei wird die Deklaration vom Typ `boolean` in Zeile 4 gemäß oben ebenfalls nicht beachtet.

```
1 {
2     int i;
3     i = 1;
4     boolean b;
5     return i;
6 }
```

## 4.2 Typisierungsfehler

Bei der Typisierung treten drei Arten von Fehlern auf. Diese werden durch die *Exceptions* im Unterpaket `exceptions` modelliert. Welche Klassen die jeweiligen *Exceptions* werfen ist in Abbildung 8 zu sehen.

**Fehlendes Symbol** Dieser Fehler tritt auf, wenn ein Variablen- oder Methodenname referenziert wird, der weder lokal noch global bereits deklariert wurde.

**Symbol wurde bereits deklariert** Deklarationen können nur bereits definierte Namen der globalen Umgebung verdecken, nicht aber bereits lokal definierte.

**Typen stimmen nicht überein** Diese Fehler treten durch Anwendung des oben besprochenen *Typenableitungssystems* auf.

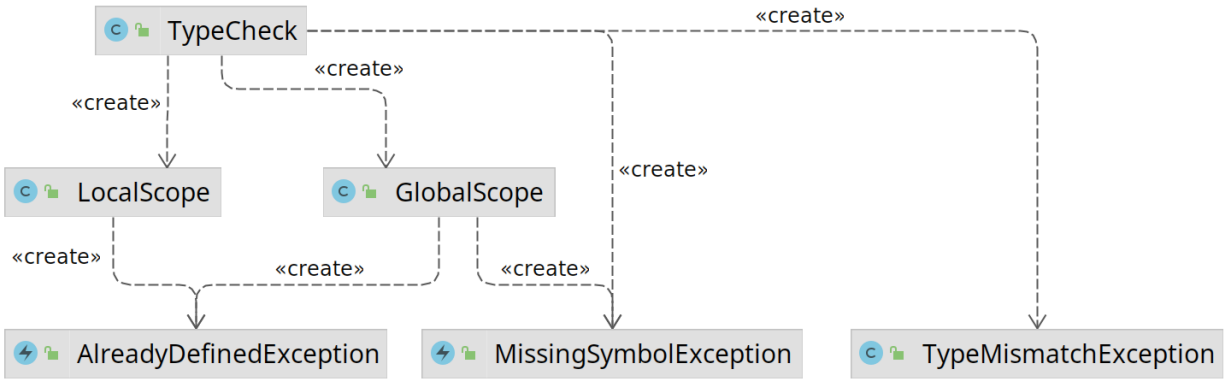


Abbildung 8: UML Diagramm des `typecheck.exceptions`-Pakets mit Klassen, welche diese *Exceptions* werfen.

# 5 Bytecode-Generator (Matthias Wallner)

Die Codegenerierung wurde mithilfe des ASM-Frameworks[3] Version 9.4 umgesetzt. Die Methode `generateProgram(Program program, String outputPath)` in der Klasse `codegen.ProgramGenerator` nimmt einen typisierten Abstract Syntax Tree und schreibt die kompilierte `.class`-Datei in das angegebenene Verzeichnis. Die Methode `codegen.ProgramGenerator.compile` akzeptiert den Pfad der zu übersetzenden Datei, sowie den Pfad zum gewünschten Output-Verzeichnis und schreibt dorthin die ausführbare Datei.

Die Codegenerierung wurde für alle Sprachelemente der in der Vorlesung definierten Untermenge von Java umgesetzt. Zusätzlich kann der Compiler noch folgenden Input übersetzen:

- direkte Initialisierung von Feldern außerhalb des Konstruktors
- das unäre Minus und die unäre Negation
- die binären arithmetischen Operatoren `+`, `-`, `*`, `/`, wobei die Addition auch für Strings definiert ist
- die binären Vergleichsoperatoren `==`, `!=`, `<`, `<=`, `>`, `>=`, wobei die Gleichheitsoperatoren auch auf Objekte angewendet werden können

# 6 Bedienungsanleitung

Dieses Projekt wurde mit `openjdk-19` Version 19.0.2 mit der Option `-enable-preview` erstellt. Diese Option ist notwendig für die Verwendung von *pattern matching* innerhalb von `switch`-Anweisungen.

Als Einsprungstelle dient die Klasse `JavaMiniCompiler`. Diese enthält eine `main`-Funktion, welche die konkrete Benutzung dieses Projekts als Compiler vereinfachen soll. Sowohl `antlr-4.11.1-complete.jar`[4] als auch `asm-9.4.jar`[5] müssen sich im *classpath* von `Java` befinden.

```
$ java --enable-preview -cp .:./antlr-4.11.1-complete.jar:./asm-9.4.jar MiniJavaCompiler
```

Deutlich handlicher ist die Benutzung dieses Projektes als *jar*-Datei, in dieser sind alle benötigten Abhängigkeiten bereits enthalten.

```
$ java --enable-preview -jar JavaMiniCompiler.jar
```

Diese Methode erhält als Argument eine Datei, welche durch den `JavaMiniCompiler` zu *Bytecode* für die `JVM` übersetzt wird. Standardmäßig werden die entsprechenden *class*-Dateien in einem Unterordner `./out/` erstellt. Dieses Verhalten kann mit der Option `-o output_dir` angepasst werden.

Optional kann eine weitere `java`-Datei als zweitletztes Argument übergeben werden. Diese wird im Anschluss mit dem regulären, sich in der Ausführungsumgebung befindlichen, *Javacompiler* übersetzt. Dies ist etwa hilfreich um eine zugehörige, ausführbare `main`-Methode zu definieren, da dies durch den `MiniJavaCompiler` nicht möglich ist. Durch die Option `-r` wird diese zusätzliche Datei nach der Übersetzung direkt ausgeführt.

```
$ java -jar --enable-preview JavaMiniCompiler.jar MatrixMain.java Matrix.java
```

Weiterhin können über die Optionen `[-aguv]` weitere Informationen auf der Konsole ausgegeben werden. `-a` gibt eine *Stringdarstellung* des typisierten Syntaxbaums aus, `-u` des untypisierten bevor die Typisierung gestartet wurde, `-g` den Inhalt der globalen Umgebung, und `-v` allgemeine Informationen über den aktuellen Fortschritt des Compilers. Ein Aufruf mit allen möglichen Optionen könnte folgendermaßen aussehen:

```
$ java -jar --enable-preview JavaMiniCompiler.jar -augrv -o . MatrixMain.java Matrix.java
```

Eine Kurzbeschreibung der Benutzung und aller vorhandenen Optionen kann mit der Option `-h` auf der Konsole ausgegeben werden.

# 7 Tests

Einige Tests, durch welche auch die Funktionalität und Korrektheit des `MiniJavaCompilers` geprüft wurden, befinden sich im Unterordner `test/` des Projekts. Eine Auswahl wird im folgenden kurz beschrieben.

`Matrix.java` Diese Datei implementiert Knoten und eine zweidimensional verkettete Liste um Matrizen darzustellen. In der `Matrix`-Klasse sind außerdem grundlegende Matrixoperationen definiert, wie die Addition, (Skalar-)Multiplikation, und die Berechnung der Transponierten. Zusätzlich existiert die Datei `MatrixMain.java` mithilfe welcher die `Matrix`-Klasse auf der Konsole getestet werden kann.

`StmtExprTest.java` Diese Datei diente dazu die korrekte Typisierung der `StatementExpressions` und der Variablendeklaration durch die Klasse `VarDeclStmt` zu testen.

`BasicClassesTest.java` Diese Datei zeigt die Funktionalität verschiedener basic Compiler-Features durch die compilierte Java-Datei `BasicClasses.java`. Durch Methoden werden Variablen Werte zugewiesen und wieder ausgelesen. Die Funktionalität einer Inline Feldinitialisierung und die korrekte Konstruktion von Costum-Klassen wird veranschaulicht. Außerdem wird gezeigt, dass Escape-Sequences in Strings richtig interpretiert werden und Strings 'addiert' werden können.

`ShapeTest.java` Diese Datei spielt mit den eigens entworfenen Klassen *Circle* und *Rectangle* und zeigt die implementierte Funktionalität aus der `ShapeClass.java` Datei.

`Examples.txt` Wir haben den Java-Code in dieser Datei mit unserem MiniJavaCompiler übersetzt und die `.class`-Dateien mithilfe des FernFlower-Decompilers rückübersetzt. Im Anhang C befindet sich ein Verleich.

# Literatur


[1] Github-Seite des MiniJavaCompilers, `https://github.com/tkpf/MiniJavaCompiler` (Stand: 02. März 2023)

[2] ANTLR (ANother Tool for Language Recognition), `https://www.antlr.org/index.html` (Stand: 1. März 2023)

[3] ASM project, `https://asm.ow2.io/index.html` (Stand: 1. März 2023)

[4] Github-Seite des ANTLR Projekts, `https://github.com/antlr/website-antlr4/tree/gh-pages/download` (Stand: 1. März 2023)

[5] Repository des ASM Projekts, `https://repository.ow2.org/nexus/content/repositories/releases/org/ow2/asm/asm/9.4/` (Stand: 1. März 2023)

# A Parse Tree Example

Der Parse-Tree für die implementierte Rectangle Klasse, der durch die MiniJava-Grammatik 1 definiert ist, resultiert in dem auf Bild 9 dargestellten Baum. Der zugehörige Java-Quellcode ist folgender:

```
class Rectangle {
    int a;
    int b;

    Rectangle (int a, int b) {
        this.a = a;
        this.b = b;
    }
    Rectangle (int a) {
        this.a = a;
        this.b = a;
    }

    int getLongerEdge() {
        if (a > b) {
            return a;
        } else {
            return b;
        }
    }

    void makeSquareWithLongerOrSmallerEdge(boolean b) {
        if (b) {
            this.a = this.b;
        }
        else {
            this.b = this.a;
        }
    }

    boolean isSquare() {
        return (this.a == this.b);
    }

}
```

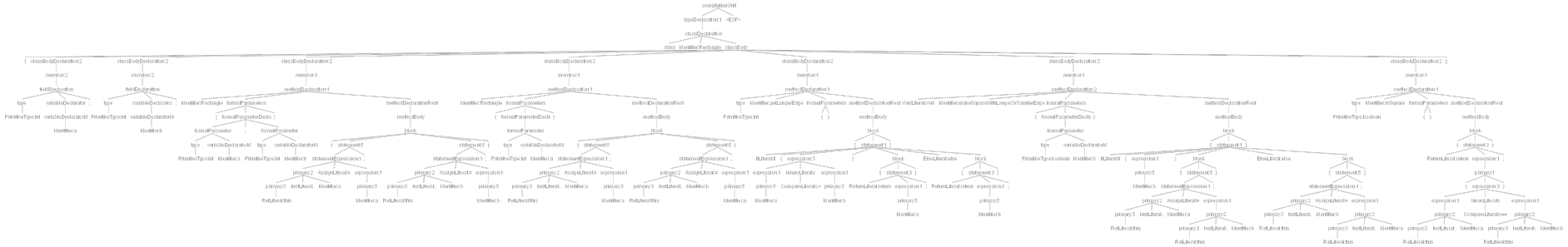

Abbildung 9: Parse Tree für implementierte Rectangle.java Klasse

# B Typenableitungsregeln

Entnommen aus den Folien der Vorlesung `INF3339c` *Spezielle Kapitel der Praktischen Informatik: Compilerbau* gehalten von Prof. Martin Plümicke an der Eberhard Karls Universität Tübingen im Wintersemester 2022/2023.

## Literal–Regeln

$$[\textbf{IntLiteral}] \quad O \rhd_{Expr} \text{Integer}(n) : \texttt{int}$$

$$[\textbf{BoolLiteral}] \quad O \rhd_{Expr} \text{Bool}(b) : \texttt{boolean}$$

$$[\textbf{CharLiteral}] \quad O \rhd_{Expr} \text{Char}(c) : \texttt{char}$$

$$[\textbf{NullLiteral}] \quad O \rhd_{Expr} \text{Null} : \theta'$$

## Expression–Regel: Simple–Expressions

$$[\textbf{Unary1}] \quad \frac{O \rhd_{Expr} e : \texttt{int}}{O \rhd_{Expr} \text{Unary}(\text{"+"}/\text{"-"}, e) : \texttt{int}}$$

$$[\textbf{Unary2}] \quad \frac{O \rhd_{Expr} e : \texttt{boolean}}{O \rhd_{Expr} \text{Unary}(\text{"!"}, e) : \texttt{boolean}}$$

$$[\textbf{Binary1}] \quad \frac{O \rhd_{Expr} e1 : \texttt{int}, O \rhd_{Expr} e2 : \texttt{int}}{O \rhd_{Expr} \text{Binary}(\text{"+"}/\text{"-"}/\text{"*"}/\text{"\%"}, e1, e2) : \texttt{int}}$$

$$[\textbf{Binary2}] \quad \frac{O \rhd_{Expr} e1 : \texttt{boolean}, O \rhd_{Expr} e2 : \texttt{boolean}}{O \rhd_{Expr} \text{Binary}(\text{"\&\&"}/\text{"||"}, e1, e2) : \texttt{boolean}}$$

## Expression–Regel: Variablen

$$[\textbf{LocalOrFieldVar}]\ \frac{O \rhd_{Id} v : \theta}{O \rhd_{Expr} \text{LocalOrFieldVar}(\, v \,) : \theta}$$

$$[\textbf{InstVar}]\ \frac{O \rhd_{Expr} re : \overline{\tau}, \quad O_{\overline{\tau}} \rhd_{Id} v : \theta}{O \rhd_{Expr} \text{InstVar}(\, re, v \,) : \theta}$$

## Statement–Regeln

$$[\textbf{Return}]\ \frac{O \rhd_{Expr} e : \theta}{O \rhd_{Stmt} \text{Return}(\, e \,) : \theta}$$

$$[\textbf{If}]\ \frac{\begin{array}{c} O \rhd_{Stmt} s_1 : \theta_1, O \rhd_{Stmt} s_2 : \theta_2 \\ O \rhd_{Expr} e : \texttt{boolean} \end{array}}{O \rhd_{Stmt} \text{If}(\, e, s_1, s_2 \,) : \overline{\theta}, \textit{wobei}\ \overline{\theta} \in \mathbf{UB}^2(\, \theta_1, \theta_2 \,)}$$

$$[\textbf{While}]\ \frac{O \rhd_{Expr} e : \texttt{boolean}, \ O \rhd_{Stmt} \text{Block}(\, B \,) : \theta}{O \rhd_{Stmt} \text{While}(\, e, \text{Block}(\, B \,) \,) : \theta}$$

[2] upper bound

## Block–Statement Regeln

$$[\textbf{BlockInit}]\quad \frac{O \rhd_{Stmt} \ stmt : \theta}{O \rhd_{Stmt} \ \text{Block}(\ stmt\ ) : \theta}$$

$$[\textbf{Block}]\quad \frac{O \rhd_{Stmt} \ s_1 : \theta,\ O \rhd_{Stmt} \ \text{Block}(\ s_2; \ldots; s_n;\ ) : \theta'}{O \rhd_{Stmt} \ \text{Block}(\ s_1; s_2; \ldots; s_n;\ ) : \overline{\theta},\ wobei\ \overline{\theta} \in \textbf{UB}(\ \theta, \theta'\ )}$$

$$[\textbf{Blockvoid}]\quad \frac{\begin{array}{c} O \rhd_{Stmt} \ s_1 : \texttt{void}, \\ O \rhd_{Stmt} \ \text{Block}(\ s_2; \ldots; s_n;\ ) : \theta \end{array}}{O \rhd_{Stmt} \ \text{Block}(\ s_1; s_2; \ldots; s_n;\ ) : \theta}$$

$$[\textbf{Block-Local-VarDecl}]\quad \frac{O\backslash\{\ v : \theta'\ \} \cup \{\ v : \overline{\theta}\ \} \rhd_{Stmt} \ \text{Block}(\ s_2; \ldots; s_n;\ ) : \theta}{O \rhd_{Stmt} \ \text{Block}(\ \text{LocalVarDecl}(\ v, \overline{\theta}\ ); s_2; \ldots; s_n;\ ) : \theta}$$

## Expression–Regel: Statement–Expressions

$$[\textbf{New}]\quad O \rhd_{Expr} \ \text{New}(\ \theta\ ) : \theta$$

$$[\textbf{Assign}]\quad \frac{O \rhd_{Expr} \ ve : \theta',\ O \rhd_{Expr} \ e : \theta}{O \rhd_{Expr} \ \text{Assign}(\ ve, e\ ) : \theta'} \quad \theta \leq^* \theta'^{1}$$

$$[\textbf{Method-Call}]\quad \frac{\begin{array}{c} O \rhd_{Expr} \ re : \overline{\tau} \\ O_{\overline{\tau}} \rhd_{Id} \ m : \theta'_1 \times \ldots \times \theta'_n \to \theta \\ \forall 1 \leqslant i \leqslant n :\ O \rhd_{Expr} \ e_i : \theta_i \end{array}}{O \rhd_{Expr} \ \text{MethodCall}(\ re, m, (e_1, \ldots, e_n)\ ) : \theta} \quad \theta_i \leq^* \theta'_i$$

[1] $\leq^*$ ist die Subtypen–Relation

# C Vergleich mit dekompliertem Code

Der zu übersetzende Java-Code:

```
    // empty classes

class emptyClass {}


// testing fields

class basicTypeFieldClass {
    int field1;
    char field2;
    boolean field3;
}

class stringFieldClass {
    String field1;
}

class Nothing {
    void doNothing(){}
}
```

```
class Store
{
    int store;

    Store(int i){
        store = i;
    }

    Store(){
        store = 0;
    }

    void countUp() {
        store = store + 1;
    }
}

// testing statements

class returnMethods{
```

Der decompilierte Java-Code:

```
public class emptyClass {
    public emptyClass() {
    }
}

public class basicTypeFieldClass {
    public int field1;
    public char field2;
    public boolean field3;

    public basicTypeFieldClass() {
    }
}

public class stringFieldClass {
    public String field1;

    public stringFieldClass() {
    }
}

public class Nothing {
    public void doNothing() {
    }

    public Nothing() {
    }
}

public class Store {
    public int store;

    public Store(int var1) {
        this.store = var1;
    }

    public Store() {
        this.store = 0;
    }

    public void countUp() {
        ++this.store;
    }
}
```

```
public class returnMethods {
```

```
    int returnFive(){
        return 5;
    }
    boolean returnTrue(){
        return true;
    }
    char returnAnA(){
        return 'a';
    }
    String returnHello(){
        return "Hello";
    }
    public emptyClass createEmpty()
    {
        return new emptyClass();
    }
}

class methodsWithInput{
    void thisMethodHasAnInput(int i) {}
}
```

```
    public int returnFive() {
        return 5;
    }

    public boolean returnTrue() {
        return true;
    }

    public char returnAnA() {
        return 'a';
    }

    public String returnHello() {
        return "Hello";
    }

    public emptyClass createEmpty() {
        return new emptyClass();
    }

    public returnMethods() {
    }
}

public class methodsWithInput {
    public void thisMethodHasAnInput(int
    ↪  var1) {
    }

    public methodsWithInput() {
    }
}
```

```
class LoopsAndIfs {
    public int choose(int a, int b,
    ↪  boolean c)
    {
        if(c) { return a; }
        else { return b; }
    }

    public int sumUp(int num)
    {
        int res = 0;
        while (0 < num)
        {
            res = res + num;
            num = num - 1;
        }
        return res;
    }

    public int sumUpRecursive(int num)
```

```
public class LoopsAndIfs {
    public int choose(int var1, int var2,
    ↪  boolean var3) {
        return var3 ? var1 : var2;
    }

    public int sumUp(int var1) {
        int var2;
        for(var2 = 0; 0 < var1; --var1) {
            var2 += var1;
        }

        return var2;
    }

    public int sumUpRecursive(int var1) {
        return var1 == 0 ? 0 :
        ↪  this.sumUpRecursive(var1 - 1)
        ↪  + var1;
    }
```

```
    {
        if (num == 0) {
            return 0;
        }
        return this.sumUpRecursive(num -
        ↪  1) + num;
    }

    public void doNothingRecursion(int
    ↪  repetitions)
    {
        if (repetitions < 1)
        {
            LoopsAndIfs l = new
            ↪  LoopsAndIfs();
            l.doNothingRecursion(repetitio┘
            ↪  ns -
            ↪  1);
        }
    }

    public void spin(int i)
    {
        while(0 < i)
        {
            i = i - 1;
        }
    }

    public int count(int i)
    {
        int counter = 0;
        while(0 < i)
        {
            i = i - 1;
            counter = counter + 1;
        }
        return counter;
    }
}

// testing constructors

class constructorClass{
    constructorClass(){}
}

class constructorClassWithThisAssignment {
    int i;
    constructorClassWithThisAssignment(int
    ↪  i) {
        this.i = i;
    }
}
```

```
    public void doNothingRecursion(int
    ↪  var1) {
        if (var1 < 1) {
            LoopsAndIfs var2 = new
            ↪  LoopsAndIfs();
            var2.doNothingRecursion(var1 -
            ↪  1);
        }

    }

    public void spin(int var1) {
        while(0 < var1) {
            --var1;
        }

    }

    public int count(int var1) {
        int var2;
        for(var2 = 0; 0 < var1; ++var2) {
            --var1;
        }

        return var2;
    }

    public LoopsAndIfs() {
    }
}

public class constructorClass {
    public constructorClass() {
    }
}

public class
↪  constructorClassWithThisAssignment {
    public int i;
```

```
class newAssignmentClass {
   constructorClass i = new
    ↪  constructorClass();
}

class newAssignmentClass2 {
   constructorClassWithThisAssignment i =
    ↪  new constructorClassWithThisAssignm
    ↪  ent(3);
}
```

```
// testing operators

class Operators {
    public String addStrings(String a,
     ↪  String b) { return a + b; }

    public boolean isSmaller(int a, int b)
    {
        return a < b;
    }

    public boolean compareObjects(Object
     ↪  a, Object b) {return a == b;}
}
```

```
// some sample classes

class Circle {
    int radius;

    Circle(int radius) {
        this.radius = radius;
    }

    int approximateCircumference() {
        return radius * 3;
    }
```

```
    public constructorClassWithThisAssignm
     ↪  ent(int var1)
     ↪  {
        this.i = var1;
    }
}

public class newAssignmentClass {
    public constructorClass i = new
     ↪  constructorClass();

    public newAssignmentClass() {
    }
}

public class newAssignmentClass2 {
    public
     ↪  constructorClassWithThisAssignment
     ↪  i = new constructorClassWithThisAs
     ↪  signment(3);

    public newAssignmentClass2() {
    }
}

public class Operators {
    public String addStrings(String var1,
     ↪  String var2) {
        return var1.concat(var2);
    }

    public boolean isSmaller(int var1, int
     ↪  var2) {
        return var1 < var2;
    }

    public boolean compareObjects(Object
     ↪  var1, Object var2) {
        return var1 == var2;
    }

    public Operators() {
    }
}

public class Circle {
    public int radius;

    public Circle(int var1) {
        this.radius = var1;
    }

    public int approximateCircumference() {
        return this.radius * 3;
```

```
    boolean guessRadius(int guess) {
        return (radius == guess);
    }

}

class Rectangle {
    int a;
    int b;

    Rectangle (int a, int b) {
        this.a = a;
        this.b = b;
    }

    int getLongerEdge() {
        if (a > b) {
            return a;
        } else {
            return b;
        }
    }

}

class someCollectionsOfObjects {
    int i = 4;
    Circle myCircle = new Circle(5);
    Rectangle myRectangle = new
    ↪  Rectangle(3, i);

    boolean checkIfRectangleWasInitialized
    ↪  Right()
    ↪  {
        if (myRectangle.getLongerEdge() ==
        ↪  i) {
            return true;
        } else {
            return false;
        }
    }

     void manipulateRadiusOfCircle(int
     ↪  manipulation) {
            if (myCircle.radius !=
            ↪  manipulation) {
                myCircle.radius =
                ↪  manipulation;
            }
        }
}
```

```
    }

    public boolean guessRadius(int var1) {
        return this.radius == var1;
    }
}

public class Rectangle {
    public int a;
    public int b;

    public Rectangle(int var1, int var2) {
        this.a = var1;
        this.b = var2;
    }

    public int getLongerEdge() {
        return this.a > this.b ? this.a :
        ↪  this.b;
    }
}

public class someCollectionsOfObjects {
    public int i = 4;
    public Circle myCircle = new Circle(5);
    public Rectangle myRectangle;

    public boolean checkIfRectangleWasInit
    ↪  ializedRight()
    ↪  {
        return this.myRectangle.getLongerE
        ↪  dge() ==
        ↪  this.i;
    }

    public void
    ↪  manipulateRadiusOfCircle(int var1)
    ↪  {
        if (this.myCircle.radius != var1) {
            this.myCircle.radius = var1;
        }

    }

    public someCollectionsOfObjects() {
        this.myRectangle = new
        ↪  Rectangle(3, this.i);
    }
}
```