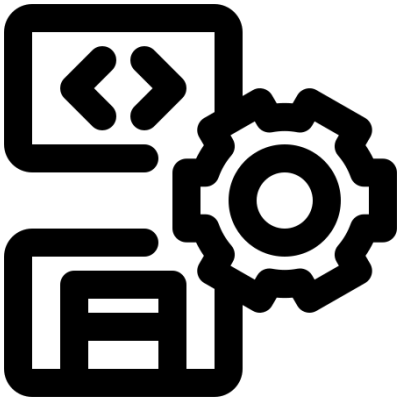


Aufbau des Compilers

MiniJavaCompiler

Ein Projekt im Rahmen der Vorlesung Compilerbau (SS 2025) an der DHBW Stuttgart.

Jonas Braun, Jonathan Kalmbach, Bernhard Mebert, Sean Reich, Tobias Schuhmacher

Gruppe: **5Compile**

Vorgestellt am: 27.06.2025

Grundstruktur

Das Compilerprojekt besteht aus 4 großen Komponenten:

1. **ASTGenerator**: Komponente, welche für den Aufbau des abstrakten Syntaxbaums zuständig ist.
2. **TypedASTGenerator**: Erzeugt aus dem abstrakten Syntaxbaum einen getypten abstrakten Syntaxbaum.
3. **ByteCodeGenerator**: Erzeugt aus dem getypten, abstrakten Syntaxbaum den Bytecode.
4. **TestSuite**: Umfangreiche Tests für die einzelnen Komponenten, sowie Integrationstests bereit.

Der Compiler an sich besteht nur aus den ersten drei Komponenten.

Teamleistungen im Überblick

Tobias Schuhmacher – Projektleitung

- Definition des initialen abstrakten Syntaxbaums (AST)
- Aufbau und Strukturierung des Projekts (GitHub, Maven, Verzeichnisstruktur)
- Koordination des Teams, Planung der Phasen
- Unterstützung bei Integration, Fehlerbehandlung und Demo

Sean Reich – Parsing & Grammatik

- Erstellung und Pflege der ANTLR-Grammatik (MiniJava.g4)
- Aufbau des ASTGenerator, der den ParseTree in den vereinbarten AST überführt

Jonas Braun – Semantische Analyse

- Aufbau der TypedAST-Strukturen (z. B. TypedClassDecl, TypedBinary, ...)
- Entwicklung des TypeCheckVisitor, inkl. symbolischer Kontextverarbeitung
- Fehlerbehandlung bei Typkonflikten und semantischer Analyse

Bernhard Mebert – Codegenerierung

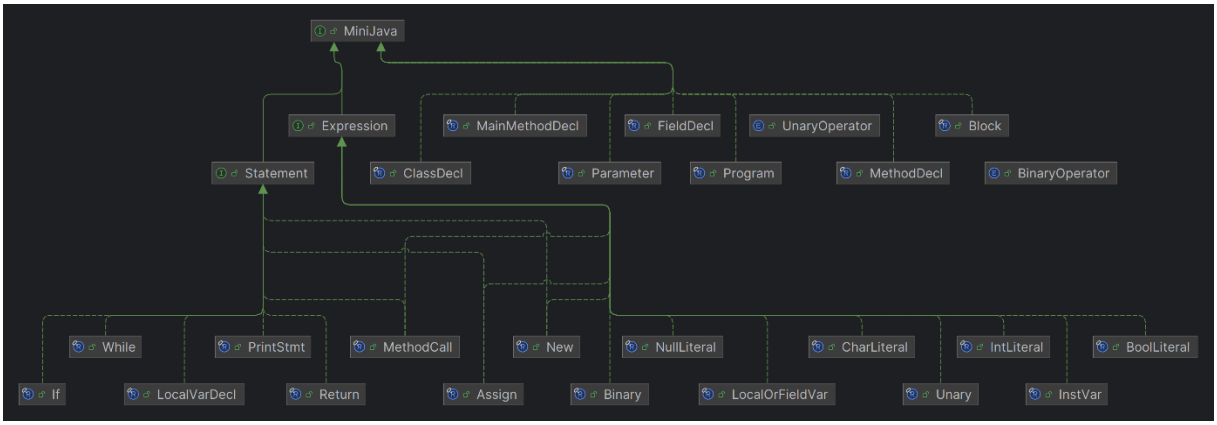
- Implementierung der codeGen()-Methoden in den Typed-Klassen
- Aufbau der Bytecode-Erzeugungslogik auf Basis des TypedAST
- Abstimmung mit Main.java zur fehlerfreien Integration

Jonathan Kalmbach – Tests & Validierung

- Entwicklung von Unit-Tests für AST, TAST und CodeGen-Komponenten
- Implementierung der ReflectionUtility
- Erstellung und Pflege der TestSuite mit Positiv- und Negativtests

AST

Zu Projektbeginn wurde ein gemeinsamer abstrakter Syntaxbaum (AST) definiert, der die unterstützten Sprachelemente festlegte und als strukturelle Basis für alle nachfolgenden Komponenten diente.



Einschränkungen des MiniJavaCompilers

- Keine Vererbung
- Nur Standardkonstruktoren erlaubt
- Keine Arrays, Threads, Exceptions, Lambda-Expressions
- Nur `int`, `boolean`, `char` als primitive Typen
- Keine Generics, Packages oder Imports
- print() statt System.out.println()
- Accessmodifier immer public
- keine arithmetische Logik (muss geklammert werden)
- kein Prekrement (--i und ++i)
- max. eine Klasse als Input (kein program)
- kein Syntactic Sugar auflösbar

Fazit

Wir haben das Projekt anfangs unterschätzt – insbesondere in technischer Hinsicht. Die Komplexität eines echten Compilers wurde uns erst im Laufe der Umsetzung vollständig bewusst. Viele Konzepte wie AST-Strukturierung, symbolische Kontexte, Typprüfung und Bytecode-Erzeugung mit ASM waren herausfordernder als erwartet.

Zudem führte die personelle Überschneidung mit einem zweiten parallelen Softwareprojekt, in dem fast dasselbe Team aktiv war, zu organisatorischen Engpässen. Die Zeit reichte oft nicht aus, um erkannte Schwächen vollständig auszubessern.

Trotzdem ist es uns gelungen, einen durchgängigen MiniJava-Compiler zu entwickeln, der alle Phasen – vom Parsing bis zur .class-Datei – abbildet. Dadurch gewannen wir ein Verständnis für Compilerarchitektur und die Möglichkeit, eine komplexe Toolchain mit ANTLR, Java, Maven und ASM in einem praktischen Projekt selbst aufzubauen.

Was wir leider nicht fertigstellen konnten:

- Anwender-Dokumentation mit geeigneten Testfällen
- Zentrale Fehlersammlung und Auswertung