Fachhochschul-Bachelorstudiengang
SOFTWARE ENGINEERING
A-4232 Hagenberg, Austria

toya
Imperative Programmiersprache für die JVM

Bachelorarbeit

zur Erlangung des akademischen Grades
Bachelor of Science in Engineering

Eingereicht von

Lukas Christian Hofwimmer

Begutachtet von FH-Prof. DI Dr. Heinz Dobler

Hagenberg, Mai 2023

software engineering
IKM Fakultät Hagenberg

Erklärung

Ich erkläre eidesstattlich, dass ich die vorliegende Arbeit selbstständig und ohne fremde Hilfe verfasst, andere als die angegebenen Quellen nicht benutzt und die den benutzten Quellen entnommenen Stellen als solche gekennzeichnet habe. Die Arbeit wurde bisher in gleicher oder ähnlicher Form keiner anderen Prüfungsbehörde vorgelegt.

Die vorliegende, gedruckte Bachelorarbeit ist identisch zu dem elektronisch übermittelten Textdokument.

Datum Unterschrift

Inhaltsverzeichnis

Kurzfassung

Toya ist eine stark typisierte, Turing-vollständige Programmiersprache für die *Java Virtual Machine* (JVM). *Toya* erlaubt die Erstellung von Funktionen, Variablen, arithmetischen Ausdrücken und Kontrollstrukturen in Form von If-Verzweigungen und For-Schleifen.

Variablen können vom Typ `int`, `double`, `string` oder `boolean` sein. Ebenso ist es möglich, Felder dieser Typen anzulegen. Die explizite Angabe des Variablentyps ist mit Ausnahme von Feldern nicht möglich. Stattdessen ermittelt *toya* anhand des zugewiesenen Ausdrucks den Typ der Variable.

Der *toya* Compiler ist in zwei Teile unterteilt: Frontend und Backend. Das Frontend kümmert sich um die Verarbeitung des Quelltextes um daraus einen abstrakten Syntaxbaum zu erzeugen. Das Backend erzeugt anschließend anhand des abstrakten Syntaxbaums Bytecode für die JVM. Als Produkt liefert *toya* class-Dateien, die die JVM verarbeitet und auswertet. Der Compiler für *toya* ist vollständig in Kotlin/JVM implementiert.

Das Frontend verwendet als zentrale Bibliothek ANTLR in der vierten Version. ANTLR ist ein Syntax-Analysator-Generator, der anhand einer Grammatik-Datei einen Syntax-Analysator in Form des *Visitor* Entwurfsmuster erzeugt. Diese Grammatik-Datei enthält die Spezifikation für *toya*. Anhand dieses generierten Syntax-Analysators durchläuft *toya* den von ANTLR erzeugten Syntaxbaum und liefert einen abstrakten Syntaxbaum, der um Typinformationen bereichert wurde. Mithilfe des Typsystems von Kotlin unterscheidet das Backend anschließend zwischen den verschiedenen Strukturen von *toya*.

Das Backend verwendet als zentrale Bibliothek ObjectWeb ASM zum Generieren von Bytecode und class-Dateien. Als erstes erzeugt *toya* eine einzelne `Main`-Klasse, in welcher alle Funktionen eines *toya* Programms liegen. Ist ein *toya* Programm über mehrere Dateien verteilt, fasst der Compiler das Ergebnis in einer class-Datei zusammen. Anschließend durchläuft das Backend den abstrakten Syntaxbaum. Jede Klasse des abstrakten Syntaxbaums besitzt eine Generierungs-Funktion, die den richtigen Bytecode für den Knoten des abstrakten Syntaxbaumes erzeugt. Sobald der abstrakte Syntaxbaum fertig durchlaufen ist, erzeugt *toya* eine ausführbare class-Datei.

Abstract

Toya is a strongly typed, Turing-complete programming language for the Java Virtual Machine (JVM). *Toya* allows the creation of functions, variables, arithmetic expressions, and primitive control flow in the form of if-branches and for-loops.

Variables can be of type `int`, `double`, `string`, or `boolean`. It is also possible to create arrays of these types. With the exception of arrays, the explicit specification of the variable type is not possible. Instead, *toya* determines the variable type based on the type of the assigned expression.

The compiler for *toya* can be divided into two parts: frontend and backend. The frontend parses the source code to create an abstract syntax tree (AST). The backend then uses the AST to generate bytecode for the JVM. *Toya* produces class files which the JVM then can process and evaluate. The compiler for *toya* is completely implemented in Kotlin/JVM.

The frontend uses the fourth version of ANTLR as its core library. ANTLR generates the parser for *toya* based on a grammar file containing the language specification. This parser uses the visitor pattern to traverse the syntax tree and parse it into an AST. The AST is enriched with type information among other things. Afterwards the backend uses this type information to differentiate between the structures of *toya*.

The backend uses ObjectWeb ASM as its core library to generate bytecode and class files. First of all, the compiler creates a single class called `Main` in which all the functions of the program reside. Even if a *toya* program is distributed over several files the compiler still only generates a single class file. The backend then traverses the AST. Each class of the AST has a `generate` function which generates the correct bytecode for a corresponding node of the AST. As soon as the backend has finished traversing the AST the *toya* compiler generates an executable class file.

Kapitel 1

Einleitung

Compiler sind ein essenzieller Grundstein der Informatik und stellen das Bindeglied zwischen Mensch und Maschine dar. Erst durch Compiler ist die Entwicklung von höheren Programmiersprachen und Programmen in einer Größenordnung möglich, die den Anforderungen des 21. Jahrhunderts gerecht werden. Ohne Compiler wäre die kommerzielle Entwicklung von Software unvorstellbar. Sie sind eine logische Konsequenz, den Anforderungen nach immer effizienter werdenden Anwendungen nachzukommen. Compiler sind die Brücke zwischen Maschinencode und dem Menschen. So weit, dass die Anweisungen an den Computer in Programmiersprachen wie SQL beinahe Englischen Sätzen entsprechen. Im Zeitalter der kollaborativen Entwicklung gilt es, Programme nicht nur zu schreiben, sondern auch für sich selbst und Kolleg:innen lesbar zu machen. All diese Aspekte laufen darauf hinaus, dass Compiler die einzige Lösung sind.

Dementsprechend ist es elementar, die Funktionsweisen und Abläufe von Compilern zu verstehen. Der beste Weg, um dies zu bewerkstelligen, ist einen Compiler selbst zu entwickeln.

1.1 Zielsetzung

Ziel dieser Arbeit ist es, eine neue Programmiersprache und einen entsprechenden Compiler dafür zu entwickeln. Dieser Compiler erzeugt Bytecode für die Java Virtual Machine, um *toya*-Programme anschließend plattformunabhängig ausführen zu können. *Toya* soll die Definition von Funktionen, primitiven Variablen und Feldern und Kontrollflüssen in Form von Verzweigungen und Schleifen erlauben. Daraus ergibt sich eine Turing-vollständige Programmiersprache, aus welcher heraus theoretisch alle anderen Programmiersprachen entstehen könnten. *Toya* orientiert sich syntaktisch an Programmiersprachen wie C und Java mit einem Fokus auf Simplizität.

1.2 Wieso die JVM?

Eine grundsätzliche Frage, die es vor der eigentlichen Arbeit zu beantworten gilt, ist ob das Kompilat des *toya* Compilers Maschinencode oder ein Zwischenprodukt in Form von Bytecode für eine virtuelle Maschine produzieren soll. Während der native Ansatz mit Maschinencode eine höhere Ausführungsgeschwindigkeit mit sich bringt, kommt der

Nachteil der Implementierungskomplexität des Compilers dazu.

Virtuelle Maschinen hingegen bieten eine Abstraktionsschicht über dem Maschinencode und machen die Entwicklung eines Compilers daher wesentlich leichter. Das ermöglicht, den Fokus auf andere Aspekte, wie die Verarbeitung des Syntaxbaumes und die Generierung des Bytecodes zu legen; was auch im Verlauf dieser Arbeit klar zu erkennen ist.

Damit ist klar erkennbar, dass eine virtuelle Maschine die Ausgabe des Compilers interpretieren soll. Unklar ist jedoch weiterhin, welche virtuelle Maschine das Ziel sein soll. Aufgrund der Menge an verfügbaren Resourcen, beschränkt sich die Auswahl auf die Java Virtual Machine und Common Language Runtime. Zweiteres bringt einige Vorteile, wie zum Beispiel, dass Typinformationen generischer Typen zur Laufzeit erhalten bleiben. *Toya* nutzt jedoch nur einen Bruchteil aller möglichen Eigenschaften, weswegen dies keine entscheidende Rolle spielt. Schlussendlich fällt die Entscheidung auf die Java Virtual Machine aufgrund der Familiarität damit. Eine Implementierung für die Common Language Runtime ist jedoch ebenso umsetzbar.

1.3 Aufbau der Arbeit

Die Arbeit beschäftigt sich zuerst mit den theoretischen Grundlagen von *toya* und den Werkzeugen, denen *toya* zugrunde liegt. Anschließend werden konkrete Implementierungsdetails präsentiert und die Funktionalität der Sprache anhand von Beispielen bewiesen.

Das Kapitel Programmiersprache toya umfasst die Spezifikation von *toya*, ohne jedoch auf Implementierungsdetails einzugehen. Anschließend kommt es im Kapitel Vergleich mit Kotlin zur Gegenüberstellung mit einer etablierten Programmiersprache. Hierbei wurde Kotlin gewählt, da es sich dabei um die Implementierungssprache des *toya* Compilers handelt. Die Kapitel Generierung des Syntaxbaums und JVM und Bytecode bieten eine theoretische Beschreibung von ANTLR und der JVM. Das Kapitel Implementierung behandelt Implementierungsdetails des Compilers und bietet einen konkreten Einblick in die Architektur anhand von Codebeispielen und Diagrammen. Tests zeigt die Funktionalität von *toya* anhand von konkreten Codebeispielen und deren Ausgaben. In Zusammenfassung, Schlüsse und Lehren wird über das Ergebnis reflektiert und weitere Schritte erläutert.

Kapitel 2

Programmiersprache toya

Toya ist eine stark typisierte, Turingvollständige Programmiersprache für die *Java Virtual Machine* (JVM) mit einem Fokus auf Simplizität. Die Syntax ist an C angelehnt. Auf die Syntax wird in diesem Kapitel bei der näheren Behandlung der einzelnen Sprachkomponenten eingegangen.

Der Name *toya*, in Anlehnung an Java und Kotlin, findet seinen Ursprung bei einer Insel im Norden Japans. Dabei handelt es sich konkret um den Kratersee 洞爺湖 (Tōya-ko), der wiederum die Insel 中島 (Nakajima) enthält. Von Tōya-ko leitet sich dann der Name *toya* ab.

Toya folgt dem imperativen Programmierparadigma. Ein *toya*-Programm besteht aus einer Menge an Funktionen und Variablen, wobei wie in Java eine `main`-Funktion zum Programmeinstieg notwendig ist. Klassen gibt es keine. Bei der Übersetzung werden aber alle Programmteile in eine `Main`-Klasse zusammengefasst, da das resultierende Kompilat mindestens eine Klasse benötigt. Sollte die `main`-Funktion nicht vorhanden sein, so kommt es zu einer Ausnahmesituation während der Syntax-Analyse.

Die Benutzer:in hat die Möglichkeit, ein *toya*-Programm auf mehrere Dateien aufzuteilen. Anschließend bei der Übersetzung sind alle `toya`-Dateien anzugeben. Unabhängig von der Anzahl an Eingangs-Dateien besteht das kompilierte Programm jedoch immer aus genau einer `class`-Datei.

2.1 Typen

Toya stellt insgesamt fünf Typen und Felder dieser Typen zur Verfügung. Diese sind `boolean`, `int`, `double` und `string`. Die Auswahl der Typen beschränkt sich auf solche, die eindeutig unterschiedliche Aufgaben erfüllen. Das Erstellen weiterer Typen ist nicht möglich.

Feld-Typen sind mit dem allgemein bekannten Suffix `[]` und dem Schlüsselwort *new* zu deklarieren. So ist zum Beispiel der Typ eines Zeichenketten-Feldes als `[]` zu schreiben.

Die JVM bietet zusätzlich noch die Datentypen `byte`, `short`, `long` und `float`. All diese Typen sind zwar für größere Programme relevant und notwendig, überschneiden sich aber und kommen in *toya* daher nicht vor. Der Sinn von *toya* ist nicht die vollständige Ausschöpfung aller JVM-Eigenschaften, sondern die explorative Implementie-

rung einer Programmiersprache. Dafür sind nicht alle Datentypen notwendig. Der Typ `returnAddress` wird hier außer acht gelassen, weil dieser nur JVM-interne Relevanz hat.

`int` hat einen Wertebereich von -2^{31} bis $2^{31} - 1$. `double` folgt der IEEE-754-Spezifikation: 1 Bit für das Vorzeichen, 11 Bit für den Exponenten und 52 Bit für die Mantisse. `boolean` hat die Werte `true` und `false`. `True` wird intern als 1 repräsentiert, `false` mit 0. Die JVM kennt keinen nativen Zeichenketten-Typ, da dieser als Referenz repräsentiert wird. Da *toya* keine Erstellung von Typen erlaubt, sind die einzigen Referenztypen Zeichenketten und Felder. Die Bytecode-Generierung von *toya* unterscheidet immer zwischen Felder und nicht-Felder, wenn es um die Auswahl der richtigen Opcodes geht. Daraus folgt, dass eine Referenz, welche kein Feld ist, immer eine Zeichenkette sein muss. Zeichenketten werden als Literale mit doppelten Anführungszeichen definiert. So ist ein Hello World String als `"Hello World"` anzugeben.

Sprachen wie Java oder Kotlin konvertieren automatisch zwischen Typen, wenn diese zum Beispiel in arithmetischen Operationen gemischt werden. So liefert eine Addition, mit einem Ganzzahl- und Gleitkomma-Operanten eine Gleitkomma-Summe. Dadurch ermöglichen diese Programmiersprachen eine gewisse Flexibilität, selbst bei statischer Typisierung. Diese automatische Konvertierung besitzt *toya* nicht. Stattdessen wird das Programm in einen Ausnahmezustand versetzt, sollten verschiedene Typen in einer Operation vorkommen.

2.2 Funktionen

Funktionen sind die zentrale Komponente von *toya* und enthalten die Programmlogik. Sie bestehen aus Funktionssignatur und Funktionskörper. Die Funktionssignatur besteht aus Funktionsname, Parameter und Rückgabewert. Der Name ist das einzig verpflichtende hierbei; Parameter und Rückgabewert sind optional. Hat eine Funktion keinen Rückgabewert, so entfallen der Pfeil und nachfolgende Typ. In Quelltext 2.1 ist eine eine einfache Funktion zu sehen, die zwei Ganzzahlen addiert und deren Summe zurückliefert.

Quelltext 2.1: Eine typische Funktion unter toya.

```
function add(lhs: int, rhs: int) -> int {
    lhs + rhs
}
```

Der Funktionskörper besteht aus einer beliebigen Folge an Anweisungen und Ausdrücken. Hat eine Funktion einen Rückgabewert, so kann mit `return` ein nachfolgender Ausdruck rückgegeben werden. Das Schlüsselwort `return` ist jedoch optional: Wenn die letzte Anweisung gleichzeitig ein Ausdruck und vom gefordertem Typen ist, dann wird automatisch dieser Ausdruck geliefert. Hierbei ist jedoch zu beachten, dass die Leserlichkeit erhalten bleibt.

Der Aufruf einer Funktion erfolgt mit dem Syntax `function <name>(parameters)`. Eine Funktion kann eine beliebig Menge an Parameter aufweisen. Für jeden Parameter kann jeder beliebige Ausdruck eingesetzt werden, solange der Formal- und Aktualtyp übereinstimmt.

2.3 Variablen

Toya erlaubt Nutzer:innen die Erstellung von Variablen in Funktionen und auf globaler Ebene. Der Syntax dafür lautet `var <name> = <ausdruck>`; die explizite Angabe eines Typs ist nicht möglich. Stattdessen leitet der Compiler anhand bekannter Typinformation des zu evaluierenden Ausdrucks den Typ ab und weist diesen Typ der Variable zu. Dieser Typ bleibt über die gesamte Lebensdauer der Variable gleich. Sobald der Typ einer Variable fixiert ist, so kann er nicht mehr geändert werden. Initialisiert man also eine Variable mithilfe eines Ganzzahl-Ausdrucks, so ist die Variable bis dessen Speicherplatz durch die automatische Speicherplatzverwaltung freigegeben wird, vom Typ `int`. Eine getrennte Deklaration und Initialisierung ist nicht möglich.

Abgesehen von typentheoretischer Relevanz bietet die Verwendung des Schlüsselwortes `var` einige Vorteile, aber auch Nachteile für Verwender:innen von *toya*. Da `var` alle anderen Typen ersetzt, erleichtert es die Schreibarbeit für Programmierer:innen ungemein. Robert C. Martin sagt jedoch in seinem nominalen Werk Martin [3] “Indeed, the ratio of time spent reading vs. writing is well over 10:1. We are constantly reading old code as part of the effort to write new code.”. Daraus folgt, dass die Lesbarkeit wichtiger als Schreibbarkeit von Code ist und hierbei zeigen sich die Schwächen. Mit nur einem Schlüsselwort kann der Typ einer Variablen nicht explizit angegeben werden und muss durch die Leser:in abgeleitet werden. Ergo ist die Lesbarkeit von *toya*-Code schlechter als bei anderen Sprachen. Um diesem Problem vorzubeugen bieten moderne Entwicklungsumgebungen Hinweise auf den Typ in der grafischen Benutzeroberfläche. Aufgrund der geringen Anzahl an Typen in *toya* ist das Fehlen von Typhinweisen jedoch vernachlässigbar.

Vergleicht man nun die Variablendeklaration und Initialisierung von *toya* 2.2 mit Java 2.3, so ist zu erkennen, dass bei der Initialisierung via Literalen die Verwendung von `var` kein Problem darstellt. Will man einer Variable den gelieferten Wert einer Funktion zuweisen, so können hier aber Schwierigkeiten hinsichtlich Schlussfolgerungen auftreten. Daher ist die bewusste und intelligente Vergabe von Variablennamen essenziell.

Quelltext 2.2: Variablendeklaration in toya

```
var number = 123
var word = "Hello World"
var value = 123.456
var bool = true
var result = someFunction()
```

Quelltext 2.3: Variablendeklaration in Java (vor Version 10)

```
int number = 123;
String word = "Hello World";
double value = 123.456;
boolean bool = true;
int result = someFunction();
```

Der Name einer Variable darf nur einmal im Gültigkeitsbereich verwendet werden, da es ansonsten zu Unklarheiten kommen kann, welche von mehreren Variablen mit dem selben Namen nun gemeint sei. Auf die Frage, was sich hinter einem Gültigkeitsbereich verbirgt, wird näher im Kapitel 6 eingegangen. Ein Name darf aus beliebig vielen Groß- und Kleinbuchstaben und Unterstrichen bestehen.

2.3.1 Felder

Ein Feld ist eine eindimensionale Folge eines bestimmten Typs mit einer fixen Länge, welche bei der Deklaration des Feldes angegeben wird und sich über die Lebensdauer des Feldes nicht mehr ändert. Zuweisungen und Deklarationen unterscheiden sich nur leicht von nicht-Feld Variablen. Der Hauptunterschied dabei besteht darin, dass die Länge bei der Deklaration und der Index bei der Zuweisung anzugegeben is.

Felder werden mit der Syntax `var <name> = new <typ>[int-Ausdruck]` deklariert. Mit `<name>[int-Ausdruck] = Ausdruck` wird ein neuer Wert auf die Speicheradresse, dies anhand des Index berechnet wird, geschrieben.

2.4 Anweisungen

Anweisungen sind Befehle, die keinen Wert liefern. Dazu gehören For-Schleifen und Return-Anweisungen.

2.4.1 For-Schleifen

For-Schleifen in *toya* verhalten sich gleich wie in vielen anderen Sprachen. Sie bestehen aus einem Schleifenkopf und einem Schleifenkörper. Der Schleifenkopf besteht aus drei Teilen von denen alle drei optional sind:

- **Zählvariable:** Eine Variablendeklaration.
- **Abbruchbedingung:** Ein Ausdruck, der zu einem booleschen Wert evaluieren muss. Solang dessen Wert `true` ist, läuft die Schleife.
- **Inkrement-Ausdruck:** Der Ausdruck, welcher nach jedem Schleifendurchlauf evaluiert wird und typischerweise die Zählvariable verändert.

Quelltext 2.4: Eine For-Schleife, die die Zählvariable auf die Konsole ausgibt.

```
for (var i = 0; i <= 10; i++) {
    println(i)
}
```

Gibt es keine Abbruchbedingung, so läuft die Schleife, solange das Programm läuft. Wie in Java kann die Entwickler:in mit der Anweisung `for (;;) { ... }` eine Endlos-Schleife erzeugen.

2.5 Ausdrücke

Alle Konstrukte, die einen Wert liefern, sind Ausdrücke in *toya*. So sind Funktionsaufrufe, die einen Rückgabewert haben, If-Verzweigungen, boolesche und arithmetische Ausdrücke, Literale und Variablen Ausdrücke. Ausdrücke evaluieren immer zu einem Wert, welcher aus dem Wertebereich eines bestimmten Typs entstammt. Abgesehen vom zwingendem Übereinstimmen des Formal- und Aktualtyps existieren keine Beschränkungen, was das Ersetzen von Ausdrücken durch andere Ausdrücke betrifft.

2.5.1 Arithmetik

Toya unterstützt Addition (`+`), Subtraktion (`-`), Multiplikation (`*`) und Division (`/`) für Ganzzahl- und Gleitkomma-Werte via Infix Notation. Bei der Evaluierung von arithmetischen Ausdrücken wird auf die Klammerung und auf die Operatorrangfolge Rücksicht genommen, sodass arithmetische Ausdrücke in der richtigen Reihenfolge evaluiert werden.

2.5.2 Boole'sche Logik

Boole'sche Ausdrücke sind essenziell für die Verwendung von If-Verzweigungen und For-Schleifen, da sie für die Zweig-Wahl, beziehungsweise als Abbruchbedingung benötigt werden. Zur Auswahl stehen die boole'schen Operatoren *Und* (`&&`) und *Oder* (`||`) und die relationalen Operatoren Größer (`>`), Größer-Gleich (`>=`), Gleich (`==`), Kleiner-Gleich (`<=`) und Kleiner (`<`). Außerdem können booleäsche Ausdrücke mit dem Präfix `!` negiert werden. Es ist zu beachten, dass relationale Operatoren eine höhere Rangigkeit als boole'sche Operatoren haben, sodass selbst ohne richtiger Klammerung der Ausdruck auf die erwartete Weise evaluiert wird. Die Rangfolge der Operatoren hierbei entspricht der von C.

2.5.3 If-Verzweigungen

If-Verzweigungen ermöglichen Entwickler:innen bedingte Ausführung zu implementieren. If-Verzweigungen sind sehr flexibel in ihrer Syntax und erlauben die Angabe von entweder einem einzelnen Ausdruck oder einem Block mit mehrere Anweisungen per Zweig.

`if (<boolean-expression>) { <statement>* } else { <statement>* }` ermöglicht die Angabe von mehreren Anweisungen pro Block, wobei jeder Block von geschwungenen Klammern umgeben ist, wie in Quelltext 2.5 zu sehen ist. Hat man die Absicht, nur einen Ausdruck pro Zweig anzugeben, dann reicht `if (<boolean-ausdruck>) expression else expression` völlig aus. Siehe Quelltext 2.6 dazu.

Quelltext 2.5: If-Verzweigung als klassische Anweisung.

```
var someBoolean = true
if (someBoolean) {
    doSomething()
    print(someBoolean)
} else {
    print(someBoolean)
}
```

Quelltext 2.6: If-Verzweigung als Ausdruck in einer Variablenzuweisung.

```
var someBoolean = false
var someInteger = if (someBoolean) 4 else 5
```

Die Stärke der If-Verzweigung in *toya* liegt darin, dass es nicht nur ein Anweisung ist, sondern auch die Verwendung als Ausdruck möglich ist. Dadurch kann eine If-Verzweigung auf der rechten Seite einer Variablenzuweisung, in arithmetischen Ausdrücken, als Bedingung in anderen If-Verzweigungen, etc. vorkommen. Andere Programmiersprachen, wie Java, bieten einen ternären Operator `var x = boolean-expression`

`? expression : expression`. Dieser existiert in *toya* jedoch nicht, da die If-Verzweigung bereits diesen Zweck erfüllt.

Um die Verwendung als Ausdruck zu ermöglichen, ist zu beachten, dass bei If-Verzweigungen mit Blöcken die letzte Anweisung ein Ausdruck sein muss. Handelt es sich um eine If-Verzweigungen mit Ausdrücken besteht jeder Zweig aus nur einem einzelnen Ausdruck. Dadurch ist diese Bedingung sowieso gegeben. Außerdem müssen die Typen in allen Zweigen übereinstimmen und der sonst optionale else-Zweig zwingend vorhanden sein. Wäre der else-Zweig nicht vorhanden, dann wäre unter Umständen kein Wert als Folge der Evaluierung gegeben.

2.5.4 Literale

Literale sind die primitivsten Ausdrücke in *toya* und stellen fixe Werte dar. Jeder Typ hat ein bestimmtes Format für Literale, sodass Typ-Inferenz möglich ist.

- **int:** Alle numerischen Werte ohne Nachkommastellen. (z.B.: `12345`)
- **double:** Alle numerischen Werte mit Nachkommastellen. (z.B.: `123.45`)
- **string:** Zeichenketten, die von doppelten Anführungszeichen umgeben sind. (z.B.: `"Hello World"`)
- **boolean:** `true` und `false`.

2.6 Kommentare

toya erlaubt die Verwendung von Kommentaren. Ein Kommentar beginnt mit einem doppelten Schrägstrich `//`. Alle weiteren Zeichen in dieser Zeile – bis zum Zeilenumbruch – ignoriert die Syntax-Analyse.

Kapitel 3

Vergleich mit Kotlin

Da der *toya*-Compiler in Kotlin impelementiert und die Syntax in einigen Aspekten auch an Kotlin angelehnt ist liegt es nahe, einen Vergleich zwischen den beiden Sprachen durchzuführen. Ein besonderer Fokus im Vergleich liegt auf der Syntax, da der generierte Bytecode zwischen den beiden Sprachen kaum zu unterscheiden ist (abgesehen von Optimierungen für individuelle Code-Stücke in Kotlin). Im Folgenden werden nur die wesentlichen Unterschiede zwischen den beiden Sprachen im Umfang von *toya* aufgezeigt. Im Sinne der Prägnanz wird über den direkten Vergleich hinaus nicht näher auf Kotlin eingegangen. Zum Vergleich mit Kotlin wird die offizielle Kotlin Dokumentation von Jetbrains verwendet [8].

3.1 Übersicht Kotlin

Kotlin ist eine seit 2011 entwickelte statisch typisierte, imperative Programmiersprache mit Elementen der funktionalen Programmimerung. Als Antwort von JetBrains auf Java und Scala bietet Kotlin eine plattformübergreifende und idiomatische Programmiersprache, welche nahtlos in das JVM Ökosystem eingegliedert ist. Neben der JVM kompiliert Kotlin auch auf JavaScript, WebAssembly und Assembly. Für letzteres verwendet Kotlin die Compiler-Infrastruktur LLVM. Das momentane Haupt-Anwendungsgebiet liegt in der mobilen Entwicklung unter Android.

Google empfiehlt seit 2019 Kotlin anstatt Java zu verwenden und bietet Teile der Standardbibliothek - Jetpack Compose - nur noch unter Kotlin an, da hierbei der Einsatz von Kotlin-spezifischen Compiler Plugins notwendig ist [7]. Andrey Breslav leitete bis 2020 die Entwicklung von Kotlin und übergab anschließend die Verantwortung an Roman Elizarov. Aktuell befindet sich Kotlin bei der Version 1.8.21.

Eines der wichtigsten Merkmale von Kotlin ist die Vermeidung des *Billion Dollar Mistake*: Null Pointer. Indem der Compiler bei der unerlaubten Zuweisung von Null-Werten in einen Ausnahmezustand versetzt wird, kann es während der Laufzeit nicht mehr zu unerwünschtem Verhalten kommen. Da es jedoch weiterhin Fälle gibt, in welchen null ein erwünschter Wert ist, kann die Entwickler:in via `?` den Typ einer Variable als *nullable* markieren.

Ein weiteres wichtiges Ziel Kotlins ist die Lesbarkeit des Codes. Dies zeigt sich vor allem in den Methoden der Standardbibliothek. So gibt es zum Überprüfen von Listen auf

deren Leerheit die Methode `isEmpty()`, aber auch dessen Negation mit `isNotEmpty()`. Diese zweite Methode ist redundant, aber erleichtert die Lesbarkeit des Codes ungemein.

Hinsichtlich der funktionalen Programmierung bietet Kotlin eine leicht verwendbare Syntax an, um zum Beispiel die Komposition von Funktionen umzusetzen. Unter anderem stehen Funktionen höherer Ordnung, Lambdafunktionen und Erweiterungsfunktionen zur Verfügung. Diese finden auch häufig Verwendung in der Standardbibliothek: *Scope functions* sind Funktionen höherer Ordnung, die auf ein Objekt eine Lambdafunktion anwenden. Konkret stehen `let`, `run`, `with`, `apply` und `also` zur Verfügung, welche sich in der Art und Weise, wie sie das Objekt behandeln, unterscheiden. Auf diese Funktionen wird nicht näher eingegangen, da diese nicht Teil dieser Bachelorarbeit sind.

3.2 Funktionen

Kotlin verwendet das Schlüsselwort `fun` zur Funktionsdefinition. Der Programmeinstieg geschieht über eine eindeutig identifizierbare `main`-Funktion, welche man optional mit einem `args: String[]` Parameter versehen kann. Entscheidet die Benutzer:in sich dazu, die `main`-Funktion parameterlos zu implementieren, erstellt der Compiler intern eine zweite `main`-Funktion mit `args: String[]` Parameter, welche dazu dient, die, von der Nutzer:in definierte `main`-Funktion aufzurufen. Wie in Quelltext 3.1 und Quelltext 3.2 zu sehen ist, ähneln sich *toya* und Kotlin, abgesehen von kleinen syntaktischen Unterschieden.

Quelltext 3.1: Funktion unter Kotlin

```
fun add(lhs: Int, rhs: Int) : Int {
    return lhs + rhs
}
```

Quelltext 3.2: Funktion unter *toya*

```
function add(lhs: int, rhs: int) -> int {
    return lhs + rhs
}
```

3.3 Variablen

In Kotlin stehen zur Deklaration von Variablen die Schlüsselwörter `var` (Wert ist veränderbar) und `val` (Wert ist unveränderbar) zur Verfügung. Die Bestimmung des Typs erfolgt entweder implizit anhand des zugewiesenen Ausdrucks oder explizit.
`val someString: String = "Hello World"` weist zum Beispiel der Variable *someString* den Wert *Hello World* zu. Die explizite Angabe des Typs `String` ist bei diesem Beispiel redundant, da aus dem Ausdruck der Typ ableitbar ist und daher nicht zwingend notwendig. Quelltext 3.3 zeigt die Initialisierung von drei Variablen mit expliziter und zwei weitere Variablen mit impliziter Typangabe in Kotlin.

Im Gegensatz dazu bietet *toya* zur Variablendeklaration nur das Schlüsselwort `var` an, da in *toya* alle Variablen veränderbar sind. Die explizite Angabe von Typen ist nicht

möglich. Der Typ leitet sich immer vom Ausdruck ab. Quelltext 3.4 zeigt die Initialisierung der selben Variablen und Werten wie in Quelltext 3.3 mit dem Unterschied, dass diese in nun in *toya* implementiert sind.

Quelltext 3.3: Variablendeklarationen in Kotlin

```
val number: Int = 123
val word: String = "Hello World"
var value: Double = 123.456
var bool = true
var result = someFunction()
```

Quelltext 3.4: Variablendeklarationen in *toya*

```
var number = 123
var word = "Hello World"
var value = 123.456
var bool = true
var result = someFunction()
```

3.4 Felder

Kotlin bietet für die Verwendung von Feldern die generische Klasse `Array` an. Da Felder, wie in anderen Sprachen, auch in Kotlin statisch in ihrer Größe sind, ist die Anzahl an speicherbaren Elementen als Konstruktorparameter anzugeben. Alternativ dazu besteht die Möglichkeit, mit der Hilfs-Funktion `arrayOf(...)` ein Feld mit Werten zu initialisieren, wie in Quelltext 3.5 zu sehen ist. Für primitive Typen existieren Klassen, *toya* hingegen beruft sich auf den C-artigen Syntax und verwendet Ausdrücke in der Form von `new <type>[int-expression]` zur Initialisierung von Feldern. Die Größe des Feldes ist statisch und als Ausdruck innerhalb der eckigen Klammern anzugeben. Das Initialisieren eines Feldes und Zuweisen auf einen Index des Feldes ist in Quelltext 3.6 zu sehen.

Quelltext 3.5: Felder in Kotlin

```
var arrayViaClass = IntArray(5)
var arrayWithValues = arrayOf(1,2,3,4,5)
arrayViaClass[2] = 5
```

Quelltext 3.6: Felder in *toya*

```
var array = new int[5]
arra[2] = 5
```

3.5 If-Verzweigung

Die Semantik und Syntax von If-Verzweigungungen in *toya* gleichen denen in Kotlin. So kann die Benutzer:in sowohl einen einzelnen Ausdruck als auch einen gesamten Programmblock als Zweig angeben. Ebenso ersetzt die normale If-Verzweigung den ternären Operator, wenn alle Zweige einen Ausdruck darstellen. Dies ist ein weiteres Mittel von Kotlin, um die Lesbarkeit des Codes zu verbessern. Quelltext 3.7 zeigt eine

If-Verzweigung, die garantiert einen Wert zurückliefert und dementsprechend dem ternären Operator gleichzusetzen ist.

Quelltext 3.7: If-Ausdruck, der sowohl in Kotlin, als auch in *toya* übersetzt

```
var number = if(3 > 4) 5 else 6
```

3.6 For-Schleifen

Während *toya* For-Schleifen ähnlich dem Stil von Java anbietet, wie in Quelltext 3.8 zu sehen ist, hat Kotlin eine deutlich kompaktere und auch flexiblere Syntax um über eine Menge zu iterieren. So hat die Benutzer:in die Möglichkeit entweder über einen `int`-Bereich oder auch über eine Enumeration zu iterieren. Ein wesentlicher Unterschied zu Java hinsichtlich For-Schleifen ist die Abwesenheit des Inkrement Operators in *toya*. Stattdessen muss die Inkrementierung mithilfe des Zuweisungs- und Additions-Operators erfolgen, zum Beispiel: `i = i+1`. Die For-Schleifen in Quelltext 3.8 und Quelltext 3.9 zeigen die Iteration über den selben Wertbereich mit einer Sprunggröße von 2.

Quelltext 3.8: Einfache For-Schleife in Kotlin

```
for (i in 0..10 step 2) {
    println(i)
}
```

Quelltext 3.9: Einfache For-Schleife in toya

```
for (var i = 0; i <= 10; i = i+2) {
    print(i)
}
```

Kapitel 4

Generierung des Syntaxbaums

ANTLR (**AN**other **T**ool for **L**anguage **R**ecognition) ist ein seit 1989 entwickeltes Werkzeug von Terrence Parr zum Erzeugen von syntaktischen und lexikalischen Analysatoren und Compilern. Die Benutzer:in definiert diese mit einer Grammatik und erzeugt daraus dann mithilfe eines von ANTLR zur Verfügung gestelltem Kommandozeilen-Werkzeug Parser und Lexer in der gewünschten Ziel-Sprache. ANTLR unterstützt unter anderem Java, C#, Python, JavaScript, Go, C++, Swift, PHP und Dart [11]. ANTLR bietet als aktuellste Version 4 an, welche große Unterschiede - allen voran der Umstieg auf eine effizientere Parsing-Methodik - zu Version 3 bietet. Der Artikel Tomassetti [12] dient als Grundlage für dieses Kapitel.

ANTLR findet auch im professionellen Umfeld Verwendung. So verwendet Twitter ANTLR zur Syntax-Analyse von über 2 Milliarden Suchanfragen pro Tag [10]. Hadoop verwendet ANTLR zur Syntax-Analyse von *Hive* und *Pig* und Netbeans analysiert den Syntax von *C++* mithilfe ANTLR [9].

Adaptive LL(*) Parsing stellt den größten Unterschied zwischen Version drei und vier von ANTLR dar. ALL(*) ist ein neuer von Terrence Parr entwickelter Parsing-Ansatz, welcher theoretisch zwar eine Laufzeitkomplexität von $\mathcal{O}(n^4)$ hat, praktisch gesehen aber lineare Performanz aufweist. Im Gegensatz zum traditionellen LL- oder LR-Parsing, das auf einer vordefinierten Grammatik basiert, analysiert ALL(*) die gesamte Eingabe, um das Parsing-Entscheidungsdiagramm zu konstruieren, das die Analyse verwendet. ALL(*) verwendet eine Technik namens adaptive Vorwärtsanalyse, um den automatisch erzeugten Parsing-Entscheidungsbaum zu verbessern. Diese Technik kombiniert Vorwärts- und Rückwärtsanalyse, um die Vorhersage der nächsten *Token* zu verbessern [5].

Der große Vorteil von ANTLR gegenüber selbst entwickelten Lösungen zum Erzeugen von Syntaxbäumen ist die Effizienz mit welcher neue Grammatikregeln definiert werden können. Diese Effizienz und Leichtigkeit in der Umsetzung hat jedoch auch seine Kosten. Da ANTLR einen umfangreichen Syntaxbaum erzeugt und es sehr unwahrscheinlich ist, dass das Programm alle Daten des Syntaxbaums benötigt, kommt es zu einem Mehraufwand für Daten, die keinen Nutzen finden.

Deswegen verwenden alle größeren Programmiersprachen (C++, Python, C#, Java, etc.) selbst entwickelte Lexer und Parser um eine substantielle Reduktion der Übersetzungszeiten zu erreichen.

Da *toya* über eine experimentelle Programmiersprache nicht hinaus geht und es den programmatischen Aufwand übersteigt, wurde aktiv gegen eine maßgeschneiderte Lösung für *toya* entschieden. Die Übersetzungszeiten unter Verwendung von ANTLR sind im Rahmen von *toya* akzeptabel. Für die Implementierung der Analyse durch reguläre Ausdrücke sprechen mehrere Gründe:

- Keine rekursive Syntax-Analyse möglich.
- Programmelemente, die an allen Stellen im Code (Kommentare zum Beispiel) auftauchen können, sind an allen potenziellen Stellen im Regex-Ausdruck zu berücksichtigen. Dies führt zu Redundanz.
- Reguläre Ausdrücke wachsen schnell und sind schwer zu verwalten. Da Programmiersprachen typischerweise auch in ihrem Funktionsumfang wachsen, sind reguläre Ausdrücke nicht dafür geeignet und führen zu einer schlechten Skalierbarkeit.

4.1 Grammatik-Definition

Die Grammatik-Definition in einer `g4`-Datei ist der Ausgangspunkt für alle weiteren Schritte in ANTLR. Diese Datei beinhaltet alle Regeln für die lexikalischen und syntaktische Analyse und Meta Informationen anhand welcher Eingabe-Datein abzuarbeiten sind. Meta Informationen befinden sich typischerweise am Beginn der Datei.

Da Leerzeichen in der Regel unwichtig sind und und keine Relevanz für die Semantik der Sprachen haben (abgesehen von Ausnahmefällen wie Python), ignoriert man diese. Dies erfolgt mithilfe der Anweisung `[ \t\n\r]+ -> skip`, welche angibt, dass Leerzeichen bei der Abarbeitung einer Eingabedatei zu überspringen sind.

Ob eine Regel den Parser oder Lexer betrifft, hängt vom Anfangsbuchstaben dieser Regel ab. Ist das erste Zeichen ein Großbuchstabe, betrifft es den Lexer; wenn nicht, den Parser. Typischerweise werden als Erstes Regeln für den Parser und als Zweites Regeln für den Lexer definiert. Die Reihenfolge der Lexer-Regeln ist von Relevanz, da in derselben Reihenfolge ANTLR diese Regeln analyisiert. Quelltext 4.1 zeigt den Aufbau einer exemplaren g4-Datei.

Quelltext 4.1: Beispielhafter Aufbau einer Grammatik-Definition für Additionen

```
grammar: addition;

// Parser Regeln
operation  : NUMBER '+' NUMBER ;

// Lexer Regeln
NUMBER     : [0-9]+ ;
WHITESPACE : [ \t\n\r]+ -> skip ;
```

4.2 Listener

Um die Ergebnisse des Syntaxbaums abarbeiten zu können, bietet ANTLR zwei Entwurfsmuster an: *Visitor* und *Listener*. Die Implementierung dieser Entwurfsmuster erzeugt die Benutzer:in mithilfe des Kommandozeilen-Werkzeug `antlr4` anhand der anzugebenden Grammatik-Datei. Außerdem gibt die Benutzer:in zusätzlich noch an, ob

entweder die Implementierung für Visitor oder Listener oder für Beide zu generieren sind. Sollen keine *Listener* generiert werden, ist dies via dem Argument `-no-listener` anzugeben.

Listener haben im Gegensatz zum *Visitor* keinen Einfluss auf den Analyse-Vorgang. Stattdessen ruft der *Tree Walker*, der den Syntaxbaum traversiert, die von ANTLR generierten Methoden für den richtigen Knotentyp anhand der Analyse-Regeln auf. Diese Methoden des *Listeners* liefern keinen Wert zurück, was die Verwaltung eines abstrakten Syntaxbaums erschwert. Aufgrund der Komplexität von *toya* sind *Listener* daher nicht empfehlenswert.

Listener bieten sich gut für Zusatzverhalten an, welches den Syntaxbaum nicht verändert. Typische Verwendungszwecke für *Listener* sind das Protokollieren von Informationen oder Ermitteln von Metadaten.

4.3 Visitor

Visitor ist ein Entwurfsmuster, welches das Ausführen eine Operation auf den Elementen einer Objektstruktur ermöglicht. Die Klassen dieser Elemente oder die Struktur selbst wird dabei nicht verändert. Das Entwurfsmuster besteht aus zwei grundlegenden Komponenten: den Elementen der Objektstruktur und dem Visitor, der die Operation auf den Elementen ausführt. Die Elemente der Struktur implementieren eine Schnittstelle mit einer Funktion, um einen *Visitor* auf das Element anzuwenden. Der *Visitor* selbst definiert Methoden für jede Klasse von Elementen, die er besuchen kann.

Um das Entwurfsmuster zu nutzen, ruft man die `accept`-Methode des *Visitors* auf dem Wurzelelement der Struktur auf, welches die Schnittstelle für die Elemente implementiert. Diese Methode wiederum ruft die entsprechende Methode im *Visitor* auf, wodurch dieser das Element besucht. Das Element gibt sich selbst als Parameter an den *Visitor* weiter, wodurch dieser auf die Eigenschaften und Methoden des Elements zugreifen und eine Operation darauf ausführen kann. Ein mögliches Problem hierbei ist, dass Fehler leichter enstehen können. Vergisst die Entwickler:in auf den Aufruf einer notwendigen `accept`-Methode, kommt es nicht zum Wechsel in einen Ausnahmezustand. Stattdessen ignoriert die syntaktische Analyse die *Token*.

Das *Visitor* Entwurfsmuster hat den Vorteil, dass es das Open-Closed-Prinzip unterstützt, da neue Operationen durch die Erstellung neuer *Visitor*-Klassen hinzugefügt werden können, ohne die existierenden Elementklassen zu ändern. Außerdem können komplexe Operationen auf der Objektstruktur durchgeführt werden, indem man mehrere *Visitor*-Klassen erstellt, die jeweils eine Teiloperation durchführen. Quelltext 4.2 zeigt die Implementierung eines *Visitors*, der Wertliterale verarbeitet.

Quelltext 4.2: Implementierung eines *Visitors* für Wertliterale.

```
class ExpressionVisitor(val scope: Scope) : toyaBaseVisitor<Expression>() {
    override fun visitValue(valueContext: toyaParser.ValueContext): Expression {
        val value = valueContext.text
        val type = TypeResolver.getFromValue(value)
        return Value(value, type)
    }
    // Rest of class...
}
```

Kapitel 5

JVM und Bytecode

Die Java Virtual Machine (JVM) ist eine von James Gosling für Sun konzipierte und in Folge von Oracle weiterentwickelte Stack-basierte virtuelle Maschine. Die JVM ermöglicht die Ausführung von Bytecode unter Linux, MacOS und Windows. Zu Beginn für die plattformunabhängige Ausführung von Java-Code entwickelt, existieren eine Vielzahl an Programmiersprachen für die JVM. Dazu gehören neben Java unter anderem Kotlin/JVM, Scala und Clojure. Als Grundlage für dieses Kapitel dient Lindholm u. a. [2].

5.1 HotSpot

Neben der Übersetzung von Bytecode auf Maschinencode beinhaltet die JVM auch den *Just-in-Time (JIT)* Compiler *HotSpot*. Sobald die Anzahl der Aufrufe einer bestimmten Methode den Schwellwert überschreitet, übersetzt *HotSpot* diese Methode in Maschinencode und ersetzt den originalen Bytecode für die restliche Ausführung des Programmes. Startet man das Programm neu, beginnt dieser Prozess von vorne. Dieser Prozess hat das Ziel, häufig verwendete Methoden zu optimieren, um dadurch die Laufzeit-Performanz zu erhöhen. Der Name stammt vom Gedanken, dass *HotSpot* heiße Regionen (engl. *hotspots*), also oft aufgerufene Methoden optimieren soll.

HotSpot bietet zwei Stufen des *JIT Compilers* an: Client (C1) und Server (C2) [4]. Der Client Compiler ist auf eine schnelle Startzeit optimiert und versucht das Programm so schnell wie möglich zu optimieren. Der Server Compiler hingegen ist auf eine hohe Leistung optimiert, weshalb das Optimierungsverfahren länger dauert, aber dafür eine höhere Programm-Performanz als Folge hat. Um nun die Vorteile von beiden Stufen ausnutzen zu können, tritt der C1 Compiler früher als der C2 Compiler in Kraft und erst sobald ein höherer Schwellwert erreicht ist, kommt es zur Optimierung durch den C2 Compiler. Die JIT-Optimierung ist in fünf Stufen aufgeteilt:

- **Stufe 0**: Die JVM nimmt keine Optimierungen vor, erhebt aber Statistiken für die Optimierung in den weiteren Stufen.
- **Stufe 1** (C1): Die JVM kompiliert triviale Methoden, erhebt in dieser Stufe aber keine Statistiken.
- **Stufe 2** (C1): Die JVM verwendet diese Stufe, um sobald wie möglich die Performanz zu erhöhen und wenn die Schlange für die C2 Optimierung voll ist. Aus

diesem Grund liegt der Methodenaufruf-Schwellwert dieser Stufe bei 0. Im weiteren Verlauf verwendet die JVM die dritte Stufe, um das Ergebnis dieser Stufe noch weiter zu optimieren.

- **Stufe 3** (C1): Dies ist die Standardstufe, welche am häufigsten in Verwendung ist. *HotSpot* optimiert Methoden anhand gesammelter Statistiken, ignoriert jedoch triviale Methoden. Der Schwellwert dieser Stufe liegt bei 2000 Methodenaufrufen.
- **Stufe 4** (C2): Dies ist die einzige Stufe, bei der der C2 Compiler zum Einsatz kommt. Die Optimierung hierbei ist aufwändiger, liefert jedoch den am höchsten optimierten Code als Ergebnis. Der Schwellwert, um diese Stufe zu erreichen, liegt mit 15 000 Methodenaufrufe am höchsten.

Entscheidet sich die Entwickler:in gegen Verwendung der gestuften Kompilierung, liegt der Optimierungs-Schwellwert bei 10 000 Methodenaufrufen. Mit dem Parameter `-XX:-TieredCompilation` besteht die Möglichkeit, gestufte Kompilierung zu deaktivieren.

5.2 Classloader

Die Aufgabe des *Classloaders* ist es, Klassen bei Bedarf dynamisch während der Laufzeit nachzuladen und zu verknüpfen. *Classloader* sind in einer Baumstruktur aufgebaut, an deren Wurzel der *Bootstrap-Classloader* steht. Dieser *Bootstrap-Classloader* lädt interne Klassen der Java Plattform und ist Ausgangsbasis für alle weiteren *Classloader*. Neben dem *Bootstrap-Classloader* gibt es standardmäßig noch den *Erweiterungs-* und *System-Classloader*. Der *Erweiterungs-Classloader* lädt Erweiterungen der primären Java Klassen. Der *System-Classloader* hat als Aufgabe, Klassen des ausgeführten Java-Programms, der *Classpath*-Umgebungsvariable und des *Classpath*-Kommandozeilen -parameters nachzuladen.

5.3 Laufzeitdatenbereiche

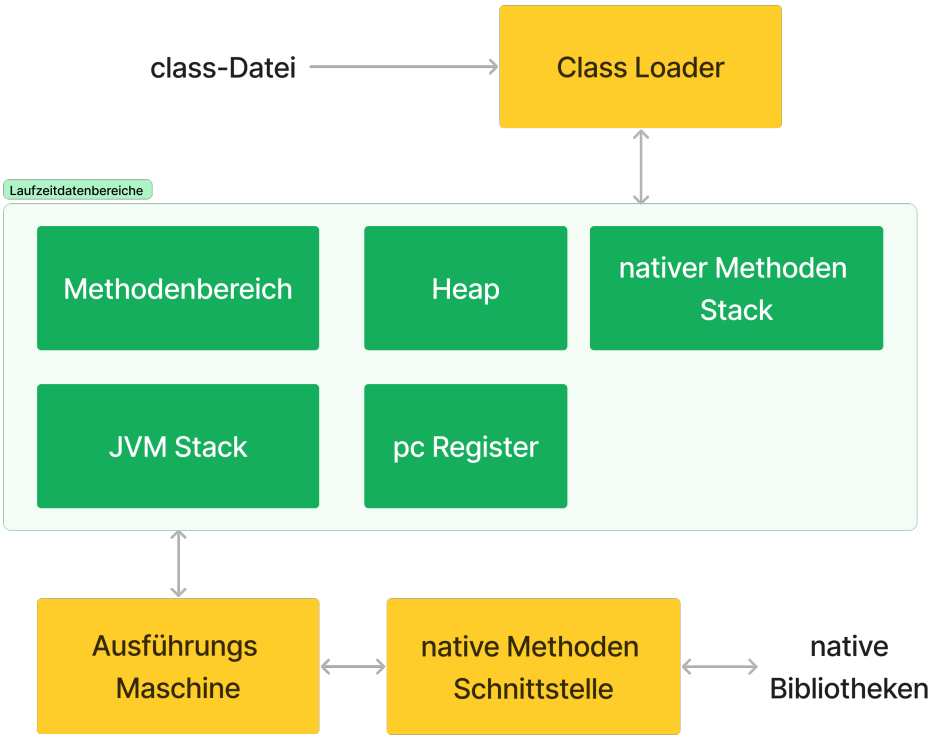
Der Laufzeitdatenbereich dient zum Speichern von Variablen, Objekten und Methoden. Ebenso umfasst es Strukturen, die Informationen über den momentanen Zustand eines Programmes enthalten. Die Architektur der JVM hinsichtlich der Laufzeitdatenbereiche ist in Abbildung 5.1 zu sehen.

Die JVM unterstützt die Erzeugung einer beliebigen Anzahl an Threads. Jeder dieser Threads besitzt ein `pc` Register, welches die Adresse der momentan ausgeführten Anweisung enthält.

Sowie jeder Thread ein `pc` Register besitzt, besitzt auch jeder Thread einen *Java Virtual Machine stack* (von hier an Stack genannt). Dieser Stack stellt einen Stapel von *Frames* dar. Frames sind Elemente variabler Größe und speichern lokale Variablen und partielle Ergebnisse. Stacks können sowohl mit einer fixen Größe, als auch dynamisch nach Bedarf der JVM definiert sein. Übersteigt die benötigte Stack-Größe die maximal erlaubte Größe, so kommt es zu einem `StackOverflowError`.

Zusätzlich zum Stack besitzt die JVM auch noch den Heap, ein Speicherbereich, den sich im Gegensatz zu `pc` Register und Stack alle Threads teilen. Im Heap liegen alle Felder und Klassen-Instanzen. Die JVM verfügt über eine automatische Speicherberei-

Abbildung 5.1: Architektur der JVM Laufzeitdatenbereiche



nigung, wenn Objekte nicht mehr benötigt werden: Der sogenannte *Garbage Collector*. Deshalb gibt es auch keine Mechanismen, Speicherplatz von Objekten freizugeben.

Im Methodenbereich liegen Strukturen, wie zum Beispiel der Bereich für Laufzeitkonstanten (*RunTime Constant Pool*), Klassenvariablen, Methoden und Inhalt der Methoden. Im Bereich für Laufzeitkonstanten liegen Wertliterale und Referenzen auf Methoden und Klassen und dient als eine Art Tabelle, aus welcher das Programm Werte referenzieren kann. Der Methodenbereich stellt einen Unterbereich des Heaps dar, muss aber im Gegensatz zum Heap nicht zwingend der automatischen Speicherbereinigung unterliegen.

Um Interoperabilität mit anderen Programmiersprachen zu gewährleisten, benötigt es den nativen Methoden Stack. Native Methoden innerhalb dieses Stacks sind unabhängig von Restriktionen der JVM, können jedoch auf deren Datenbereiche zugreifen. Kommt es zum Aufruf einer nativen Methode, so wechselt die JVM vom herkömmlichen Stack zum nativen Methoden Stack, führt diese Methode aus und liefert, wenn vorhanden, ein Ergebnis zurück.

5.4 Bytecode

Da die JVM nicht direkt den Code von JVM-basierten Programmiersprachen lesen kann, benötigt es eine Zwischensprache, den Bytecode. Dieser Bytecode entsteht bei der Übersetzung von JVM-basierten Programmiersprachen. Die class-Dateien, die bei der Übersetzung des Quelltextes erzeugt werden, enthalten als Resultat diesen Bytecode innerhalb der Methodenrümpfe.

Eine Anweisung des Bytecodes besteht aus einem Opcode, auch *mnemonic* genannt, gefolgt von keinem oder mehr Operanden. Oft folgen dem Opcode keine Operanden. Quelltext 5.1 zeigt die illustrative Schleife, die den Bytecode interpretiert, [2, siehe S. 25].

Quelltext 5.1: Auszug aus der JVM Spezifikation, welche die Interpretationsschleife für Bytecode repräsentiert.

```
do {
    atomically calculate pc and fetch opcode at pc;
    if (operands) fetch operands;
    execute the action for the opcode;
} while (there is more to do);
```

Opcodes sind für die verschiedenen Typen der JVM separat implementiert. Will die Entwickler:in zum Beispiel zwei Variablen vom Typ `int` und `double` laden, um diese später zu addieren, müssen diese im ersten Schritt mit den beiden Opcodes `iload` und `dload` geladen werden. Nach dem Laden der beiden Werte liegen diese nun am Operanden-Stack.

Einige Opcodes, `iload` unter anderem, haben zusätzliche Varianten, die einen Suffix im Format `_<number>` enthalten. Das Verhalten dieser Opcodes unterscheidet sich nicht von den suffixlosen Varianten, sondern dient als Speicherplatzsparmaßnahme. Die Zahl am Ende des Suffixes stellt den Index im lokalen Variablenfeld dar, auf das der Opcode zugreift. Dadurch vermeidet die JVM die Notwendigkeit eines zusätzlichen Parameters für häufig verwendete Indizes und spart ein Byte Speicherplatz. Die Anzahl an Varianten ist je nach Opcode unterschiedlich. `iload` bietet zum Beispiel `iload_0` bis `iload_3`.

Da es sich dabei um zwei verschiedene Typen handelt, konvertiert man den kleineren Typ (in diesem Fall `int`) zum größeren Typ mit dem Opcode `i2d`. Vom Namen lässt sich ableiten, dass diese Anweisung die Ganzzahl zu einer Gleitkommazahl konvertiert. Wichtig hierbei ist, dass der zu konvertierende Wert oben am Operanden-Stack aufliegen muss, da die Operation `i2d` den obersten Wert des Operanden-Stack entnimmt und den konvertierten Wert anschließend zurücklegt.

Mit zwei Gleitkommawerten kann man nun anschließend `dadd` aufrufen. Diese Operation entnimmt die beiden obersten Werte dem Operanden-Stack, errechnet die Summe und legt diese anschließend wieder auf den Operanden-Stack.

All diese Operationen sind für alle weiteren Typen der JVM definiert und sind der Spezifikation zu entnehmen. Die Größe aller Opcodes ist zugunsten der Kompaktheit auf ein Byte beschränkt. Daraus ergibt sich eine maximale Anzahl von 256 Opcodes.

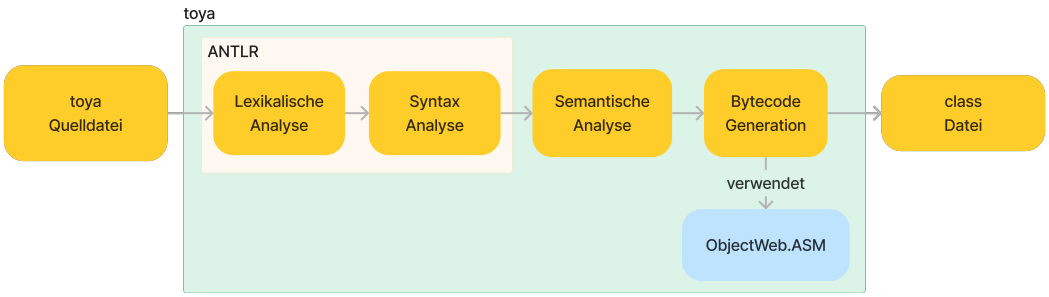
Kapitel 6

Implementierung

Die Implementierung des Compilers ist in zwei Teile aufgeteilt: Frontend und Backend. Die Kommunikation zwischen den beiden Teilen erfolgt mithilfe eines abstrakten Syntaxbaums (AST - *abstract syntax tree*), welchen das Frontend erzeugt. Dieses Kapitel geht auf die Architektur und Implementierung dieser Teile ein. Abbildung 6.1 zeigt die Archtektur des *toya* Compilers. Auf die genauere Implementierung der verschiedenen Schritte des Compilers geht dieses Kapitel ein.

Die Implementierung erfolgt in Kotlin, was das Einbinden von bereits existierenden Bibliotheken des JVM Ökosystems problemlos ermöglicht. Konkret geht es dabei um ANTLR und ObjectWeb ASM, welche wichtige Rollen im Compiler übernehmen. Die Möglichkeit, Summentypen in Form von *sealed* Schnittstellen und Klassen implementieren zu können, erleichert die Abarbeitung des ASTs wesentlich.

Abbildung 6.1: Compiler-Architektur



6.1 Frontend

Im Frontend erfolgt die lexikalische, syntaktische und teilweise die semantische Analyse. Die lexikalische Analyse zerlegt anhand der bereitgestellten Grammatik den Quellcode in *Token*. Anhand dieser *Token* erstellt die syntaktische Analyse den Syntaxbaum, welcher als Basis für den abstrakten Syntaxbaum dient.

6.1.1 Grammatik

Als Ausgangspunkt des Frontends dient die Grammatik, anhand welcher ANTLR die lexikalische und syntaktische Analyse implementiert. Diese Grammatik definiert die Compiler-Bauer:in in einer `g4`-Datei. Leerzeichen haben keine semantische Relevanz, weshalb *toya* diese in der syntaktischen Analyse überspringt. Als *Token* definiert *toya* folgende Zeichen:

- Die Schlüsselwörter `match`, `for`, `for`, `if`, `else`, `var`, `true`, `false` und `null`
- Die Infix-Operatoren `>`, `>=`, `<`, `<=`, `==`, `!=`, `&&`, `||` und der Präfix-Operator `!`
- Die arithmetischen Operatoren `+`, `-`, `*` und `/`

Beim Versuch, ein Schlüsselwort als Variablenname zu Verwenden tritt der Ausnahmezustand `VariableNameIsKeywordException` ein.

6.1.2 Abarbeitung des Syntaxbaum

Wie bereits in Generierung des Syntaxbaums erläutert, bietet ANTLR die Entwurfsmuster *Visitor* und *Listener* an, um den Syntaxbaum zu traversieren. *Toya* verwendet das *Visitor*-Muster aufgrund des Umfangs des zu traversierenden Syntaxbaums. Das Resultat des *Visitors* ist ein AST mit dem Wurzelelement `Compilation`. Diese Klasse speichert die Funktionssignaturen, globale Variablen und ein globales `Scope`, welches sich über das ganze Programm erstreckt. Alle weiteren `Scopes` nehmen dieses globale `Scope` als Grundlage.

Die wichtigste Methode des *Visitor*, welche die Abarbeitung des Syntaxbaums überhaupt ermöglicht ist die `accept` Methode. Diese Methode benötigt als Parameter einen *Visitor* vom generischen Typ `ParseTreeVisitor`. *Toya* implementiert folgende *Visitor*:

- CompilationVisitor
- StatementVisitor
- BranchVisitor
- CompositeVisitor
- ExpressionVisitor
- FunctionSignatureVisitor
- FunctionVisitor

Um das Traversieren der Knoten des Syntaxbaums zu ermöglichen, generiert ANTLR für alle Regeln der Grammatik-Definition Methoden im Format `visit<RuleName>(ctx: toyaParser.<RuleName>Context)`. Jede dieser Methoden liefert einen Wert vom generischen Typ des `toyaBaseVisitor` zurück. Quelltext 6.1 zeigt die `visit`-Funktion für Variablendeklarationen.

Quelltext 6.1: *Visitor*-Funktion zum Erstellen eines `VariableDeclarationStatement`

```
class StatementVisitor(val scope: Scope) : toyaBaseVisitor<Statement>() {
    override fun visitVariableDeclaration(ctx: toyaParser.VariableDeclarationContext
    ): Statement {
        val varName = ctx.name().text
        if(varName.isReservedKeyword()) throw VariableNameIsKeywordException(varName
    )
        val expression = ctx.expression().accept(ExpressionVisitor(scope))
```

```
            scope.addLocalVariable(LocalVariable(varName, expression.type))
            return VariableDeclarationStatement(varName, expression)
        }
        // Rest of class
    }
```

Zur Behandlung von Syntaxfehler bietet ANTLR die Klasse `BaseErrorListener` um der Benutzer:in relevante Informationen anzuzeigen. *Toya* zeigt mithilfe dieser Klasse an, welche Anweisung in welcher Zeile und welches Zeichen die syntaktische Analyse verhindert.

6.1.3 Typ-System

Die Basis für das Typ-System in *toya* ist die Schnittstelle `Type` und die davon abgeleitete Enum-Klasse `BasicType`, welche alle in *toya* verfügbaren Typen umfasst. *Toya* erlaubt keine explizite Definition des Typs einer Variable, weshalb der Typ immer vom Wert der Variable abzuleiten ist. Zum ermitteln des Typs kommt die Klasse `TypeResolver` zum Einsatz. Anhand von regulären Ausdrücken ermittelt dieser den Typ des Wert-Literals, wie in Quelltext 6.2 zu sehen ist.

Quelltext 6.2: Methode zur Ermittlung des Typs bei Wert-Literalen

```
fun getFromValue(value: String?): Type {
    if (value.isNullOrEmpty()) return BasicType.VOID
    if (isBoolean(value)) return BasicType.BOOLEAN
    if (isDouble(value)) return BasicType.DOUBLE
    if (isInt(value)) return BasicType.INT
    if (isString(value)) return BasicType.STRING
    throw UnableToInferTypeException(value)
}
```

Da nun die jeder Ausdruck Typinformationen besitzt, ermöglicht das die semantische Analyse von Ausdrücken. Die Erweiterungsfunktion `Type.checkTypeMatch` überprüft, ob beide Operanden vom selben Typ sind. Wenn nicht, kommt es zum Ausnahmezustand `BinaryOperationTypeMismatchException`. Die Implementierung für `Type.checkTypeMatch` ist in Quelltext 6.3 zu sehen.

Quelltext 6.3: Methode zur Überprüfung, dass Operanden übereinstimmende Typen haben

```
fun Type.checkTypeMatch(rhs: Type) {
    if (this != rhs) throw BinaryOperationTypeMismatchException(this, rhs)
}
```

6.1.4 Standardfunktionen

Standardfunktionen sind Funktionen, welche jedem *toya*-Programm ohne Weiteres zur Verfügung stehen. Standardfunktionen sind kein Bestandteil der lexikalischen Analyse. Die Funktion `visitFunctionCall` des `ExpressionVisitors` prüft mithilfe der Funktion `isStandardFunction`, ob es sich bei einer Funktion um eine Standardfunktion handelt. Ist dies der Fall, erzeugt der *Visitor* kein Objekt vom Typ `FunctionCall`, sondern ein Objekt, welches von `StandardFunction` erbt. Im Fall von `print` ist das zum Beispiel `PrintFunction`.

Standardmäßig bietet *toya* die Standardfunktion `print` an, welche einen Aufruf der Java Funktion `System.out.println` durchführt. Die Architektur des *toya*-Compilers ist darauf ausgelegt, weitere Standardfunktionen hinzufügen zu können. Besteht dieser Wunsch, sind an folgenden Stellen von Entwicklerseite her Veränderungen vorzunehmen:

- Im `when` innerhalb der `ExpressionVisitor.visitFunctionCall` Funktion muss ein Fall für die zu implementierenden Funktion hinzugefügt werden.
- Innerhalb der `StandardFunctions.kt` Datei: In der `standardFunctions` Liste ist eine FunktionsSignatur der neuen Standardfunktion hinzuzufügen und eine Klasse, welche von `StandardFunction` und `Expression` erbt, zu definieren.
- In der `StandardFunctionGenerator` Klasse ist der zu generierende Bytecode zu definieren.

6.2 Abstrakter Syntaxbaum

Die explizite Umwandlung auf einen abstrakten Syntaxbaum ist theoretisch nicht immer nötig. *Toya* verwendet aber ANTLR und der daraus resultierende Syntaxbaum enthält teilweise zu viele Informationen. Teilweise fehlen auch notwendige Informationen, wie zum Beispiel die über den Typ eines Ausdrucks. Daher macht es Sinn, diesen Syntaxbaum auf einen AST umzubauen.

Ein großteil des AST behandelt die Einteilung von Ausdrücken und Anweisungen in ein granulareres Klassen-Schema. Die Einteilung in zum Beispiel `LessEqualExpression` und `ForStatement` anstatt die bloße Gliederung in `Expression` und `Statement` ermöglicht eine gut skalierbare und verständliche Lösung für die Bytecode-Generierung im Backend. Die Klassenhierarchie für `Statement` und `Expression` ist in Abbildung 6.2 zu sehen. Schnittstellen und abstrakte Klassen sind respektive grün und orange markiert.

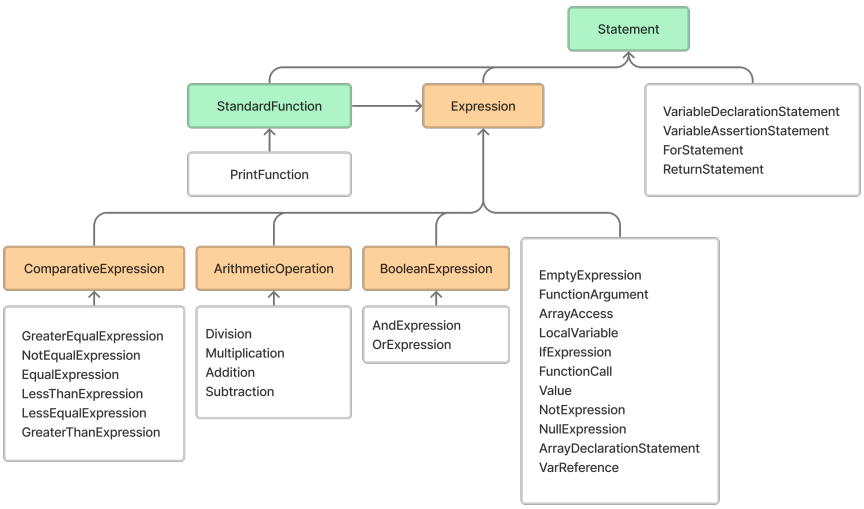
Kotlin bietet zur kompakten Erstellung von Datenobjekten sogenannte *data classes* an. `data class` Objekte erzeugen für Mitglieder des primären Konstruktor folgende Funktionen:

- `equals()` und `hashCode()`.
- `toString()` im Format `Addition(left=42, right=31)`.
- `componentN()`, die bei der Destrukturierung von Objekten verwendet werden. Hierbei ist die Reihenfolge der Definition relevant.
- `copy()` zum durchführen einer tiefen Kopie des Objekte.

Toya verwendet *data classes* für fast alle Klassen des AST. Ausgenommen sind Basisklassen, von denenen andere Klassen ableiten, da die Vererbung von *data classes* zu Problemen mit dem Verhalten von *equals()* zum Beispiel führt. Java bietet seit Version 14 `record` als Äquivalent zu diesem Konzept an. Jedoch fehlen bei `record` die `copmonentN()` und `copy()` Funktionen.

Die `Scope` Klasse ist eine zentrale Komponente in der Verwaltung des AST und wird unter anderem für die semantische Analyse benötigt. Sie speichert die lokalen Variablen für einen gegebenen Block und die Methodensignaturen eines *toya* Programms. Bei Operationen, deren Operanden lokale Variablen sind, überprüft *toya* mithilfe von `Scope`, ob diese Variablen im momentanten Kontext zur Verfügung stehen.

Abbildung 6.2: AST-Architektur für Ausdrücke und Anweisungen.



Im Zuge der Bytecode-Generierung ermittelt *toya* mithilfe `Scope` den Index einer lokalen Variable. Hierbei reicht es nicht aus, den Index der Variable in der Liste `localVariables` zu ermitteln. Variablen vom Typ `double` und `long` benötigen zwei Plätze im *Run-Time Constant Pool*, da deren Indizes 16 Bit, anstatt der üblichen acht Bit einnehmen. Da *toya* den Typ Double implementiert, ist diese Eigenschaft zu berücksichtigen. Diese Berechnung erfolgt durch eine Reduktions-Operation in der Funktion `getLocalVariableIndex`, wie in Quelltext 6.4 zu sehen ist. Diese Reduktions-Operation summiert alle Indizes bis inklusive dem Index der gesuchten Variable auf und addiert pro Index eins, beziehungsweise zwei, wenn die Variable vom Typ `double` ist.

Quelltext 6.4: Ermittlung des Index einer Variable in einem `Scope`

```
fun getLocalVariableIndex(varName: String) : Int {
    // hotfix for handling 2-wide index types (double, long, etc)
    return localVariables
        .subList(0, localVariables.indexOf(getLocalVariable(varName)))
        .fold(0) { acc, next ->
            acc + if (next.type == BasicType.DOUBLE) 2 else 1
        }
}
```

`Scope` verwaltet nicht nur Variablen, sondern auch Funktionssignaturen. Beim Aufruf einer Funktion, überprüft `Scope`, ob diese Funktion auch definiert ist. Wenn nicht, tritt der Ausnahmezustand `MethodSignatureNotFoundException` ein. Während das Überladen von Funktionen erlaubt ist, können keine Funktionen mit identischer Signatur existieren, da diese in einem `Set` gespeichert sind.

6.3 Backend

Das Backend hat als zentrale Aufgabe die Code-Generierung anhand des abstrakten Syntaxbaums, den das Frontend erzeugt hat. Zum Erzeugen des Bytecodes verwendet das Backend ObjectWeb ASM als zentrale Bibliothek. Als Resultat liefert das Backend ein Byte-Feld, das anschließend ein `FileOutputStream` in eine class-Datei schreibt.

6.3.1 ObjectWeb ASM

ObjectWeb ASM ist eine Bibliothek zum Lesen, Bearbeiten und Erzeugen von Bytecode für die JVM. Sie bietet eine Schnittstelle, um Funktionen, Klassen und einzelne Anweisungen zu erzeugen. *Toya* verwendet ASM zum Erzeugen einer Klasse und Funktionen anhand des AST. Neben der Bytecode-Generierung erfolgt im Backend der Teil der semantischen Analyse für welchen Informationen des AST notwendig sind. ASM ist kein Akronym sondern eine Anlehnung an das Schlüsselwort `asm` in der Programmiersprache C [1].

Da *toya* keine Definition von Klassen erlaubt, reicht es aus, eine statische Klasse, unabhängig vom eigentlichen Quelltext zu generieren. Version dieser Klasse ist 52, was Java 8 entspricht. Einen höhere Version ist nicht nötig, da alle Bestandteile von *toya* sehr primitiv sind.

Klassen erstellt ASM mithilfe des `ClassWriter`. Der Konstruktor dieser Klasse hat einen Parameter `flags`. Im Falle von *toya* ist dies `COMPUTE_FRAMES + COMPUTE_MAXS`. `COMPUTE_MAXS` und `COMPUTE_FRAMES` ermöglicht die automatische Berechnung der maximal erlaubten Anzahl an lokalen Variablen, die maximale Größe des Stacks und die Berechnung aller *Stack Map Frames*. Siehe dazu Quelltext 6.5.

Quelltext 6.5: Erstellung einer Klasse mithilfe ObjectWeb ASM

```
val classWriter: ClassWriter = ClassWriter(
    ClassWriter.COMPUTE_FRAMES + ClassWriter.COMPUTE_MAXS
)
classWriter.visit(
    CLASS_VERSION,
    Opcodes.ACC_PUBLIC,
    className,
    null,
    "java/lang/Object",
    null
)
```

Zum Erzeugen von Methoden und und deren Logik bietet ObjectWeb ASM die Klasse `MethodWriter`. Diese Klasse ermöglicht das Schreiben der Bytecode Anweisungen innerhalb einer Methode. Außerdem ermöglicht `MethodWriter` unter anderem auch das Setzen von *Labels* um Sprung-Befehle durchführen zu können. Sprung-Befehle sind für den Kontrollfluss wichtig, um in If-Verzweigungen nicht zutreffende Zweige zu überspringen und um bei For-Schleifen an den Beginn der Schleife zurückzukehren. Quelltext 6.6 zeigt die `generate`-Funktion zum Generieren von Wertliteralen mithilfe von ObjectWeb ASM.

Quelltext 6.6: `generate()` Funktion, welche Wert-Literale erzeugt.

```
private fun generate(value: Value) {
```

```
        val type = value.type
        val stringValue = value.value

        type.handleTypeGroups(
            i = {
                val intValue = stringValue.toInt()
                methodVisitor.visitLdcInsn(intValue)
            },
            d = {
                val doubleValue = stringValue.toDouble()
                methodVisitor.visitLdcInsn(doubleValue)
            },
            a = { methodVisitor.visitLdcInsn(stringValue.trim('"')) },
            z = {
                val opcode = if (stringValue == "true") Opcodes.ICONST_1 else Opcodes.
    ICONST_0
                methodVisitor.visitInsn(opcode)
            }
        )
    }
```

6.3.2 Summentypen

Im Backend kommen Summentypen in Form von *sealed classes* zum Einsatz. *Sealed classes* in Kotlin besitzen die besondere Eigenschaft, dass alle Kindklassen zur Übersetzungszeit vollständig bekannt sind. Dies ermöglicht zum Beispiel die Verwendung von erschöpfenden `when`-Ausdrücken in Kombination mit Polymorphismus. Kindklassen einer *sealed class* sind alle im selben Modul zu definieren. Neben Klassen können auch Schnittstellen als `sealed` markiert sein. Ist eine Klasse als *sealed* markiert, ist diese Klasse zusätzlich implizit eine abstrakte Klasse.

Wichtige *sealed classes* und *sealed interfaces* des abstrakten Syntaxbaums sind folgende:

- `ArithmeticOperation`: Umfasst Ausdrücke Addition, Subtraktion, Multiplikation und Division.
- `BooleanExpression`: Umfasst Ausdrücke für die logischen Operatoren `Und` und `Oder`. Eine `BooleanExpression` liefert immer einen boole'schen Wert zurück.
- `ComparativeExpression`: Umfasst Ausdrücke für Größer, Größer-Gleich, Gleich, Kleiner, Kleiner-Gleich und Ungleich. Eine `ComparativeExpression` liefert immer einen boole'schen Wert zurück.
- `Branch`: Umfasst die beiden Möglichkeiten, Verzweigungen im Kontrollfluss entweder als Anweisungsblock oder als einzelnen Ausdruck zu definieren.

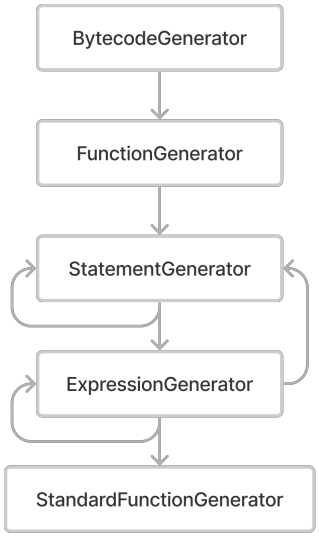
6.3.3 Architektur der Code-Generierung

Der Ablauf der Code-Generierung ist auf fünf Klassen aufgeteilt, wie in Abbildung 6.3 zu sehen ist. Als Einstiegspunkt dient immer die Klasse `ByteCodeGenerator`. Diese Klasse erzeugt eine öffentliche Klasse, die von `Object` erbt. Als nächstes ruft `ByteCodeGenerator` für jede Funktion die `generate`-Methode des `FunctionGenerator` auf.

Der `FunctionGenerator` iteriert über die Liste aller Anweisungen der Funktion und ruft für jede Anweisung die `generate`-Funktion des `StatementGenerator auf`. Nach dem durchlaufen der Anweisungsliste überprüft der `FunctionGenerator`, ob die Funktion als letzte Anweisung eine Rückgabeanweisung definiert und ob die Funktion überhaupt einen Wert zurückliefert. Liefert nun die Funktion einen Wert zurück, hat aber keine Rückgabeanweisung an letzter Stelle, ist diese Anweisung zu generieren. Anhand des Typs der letzten Anweisung der Funktion wählt der `FunctionGenerator` nun den richtigen Opcode aus und generiert die Anweisung mithilfe von ObjectWeb ASM. Ist die letzte Anweisung zum Beispiel eine Ganzzahl, generiert der *toya*-Compiler den Opcode `ireturn`.

Der `StatementGenerator` generiert entweder eine Anweisung selbst oder delegiert diese Aufgabe an den `ExpressionGenerator`, wenn es sich bei der Anweisung um einen Ausdruck handelt. Konkret generiert der `StatementGenerator` Bytecode für Variablen- und Felddeklarationen, Variablenzuweisungen, explizite Rückgabeanweisungen und For-Schleifen. Ein Großteil der Komplexität in dieser Klasse stammt daher, dass verschiedene Opcode für die verschiedenen Typen von *toya* zu beachten sind.

Abbildung 6.3: Ablauf der Code-Generierung im Backend.



Um den Mehraufwand für die Berücksichtigung jedes Typs so gut wie möglich zu minimieren, sind die Erweiterungsfunktionen `Type.handleTypeGroup()` und `Type.handleTypeArrays()` definiert, die Implementierung und Verwendung von letzteres ist in Quelltext 6.7 zu sehen. Diese beiden Funktionen höherer Ordnung benötigen eine Funktion für jeden Typ als Parameter. Dadurch liegt in jeder Funktion nur der Code, der für den entsprechenden Typ relevant ist.

Ebenso ist auch die Ausnahmebehandlung mithilfe dieser Funktionen umsetzbar. Versucht man zum Beispiel zwei Referenzen oder boole'sche Ausdrücke zu addieren, tritt der Ausnahmezustand `UnsupportedOperationOnTypeException` ein. Dadurch, dass das Verhalten für jeden Typ verpflichtend definiert sein muss, vermeidet die Entwickler:in auch, dass das Verhalten für einen Typ ungeklärt bleibt.

Quelltext 6.7: Die Erweiterungsfunktion `Type.handleTypeArrays()`, um den richtigen Opcode zum Speichern einer Variable zu ermitteln.

```
fun <T> Type.handleTypeArrays(
    ia: () -> T,
    da: () -> T,
    aa: () -> T,
    ba: () -> T
): T {
    return when (this) {
        BasicType.INT_ARR -> ia()
        BasicType.DOUBLE_ARR -> da()
        BasicType.STRING_ARR -> aa()
        BasicType.BOOLEAN_ARR -> ba()
        else -> throw NotImplementedError("handling for type '\${this.typeName}' not
     implemented")
    }
}

val opcode = localVariable.type.handleTypeArrays(
    ia = { Opcodes.IASTORE },
    da = { Opcodes.DASTORE },
    aa = { Opcodes.AASTORE },
    ba = { Opcodes.BASTORE }
)
```

Der `ExpressionGenerator` generiert Ausdrücke oder delegiert Standardfunktionsaufrufe an den `StandardFunctionGenerator`. Ist der konkrete Typ des Ausdrucks `PrintFunction`, delegiert der `ExpressionGenerator` die Bytecode-Generierung an den `StandardFunctionGenerator`. Konkret generiert der `ExpressionGenerator` Bytecode für folgende Ausdrücke:

- Variablenaufrufe
- Funktionsaufrufe
- Funktionsargumente
- Wertliterale
- arithmetische Operationen
- Vergleichsoperationen
- Logikoperationen
- Null-Ausdrücke
- Zugriff auf Felder
- If-Ausdrücke

Der `StandardFunctionGenerator` generiert alle Standardfunktionen. Im Falle der `PrintFunction` verwendet *toya* die `System.out.println` Implementierung von Java. In einem ersten Schritt ermittelt *toya* die Referenz des `PrintSteam` Typs in Form der statischen Variable `out` und evaluiert *toya* den Wert des Ausdrucks, der auszugeben ist. Anschließend wird die `println` Funktion des `out` Objekts aufgerufen. Die `println` Funktion verwendet den oben aufliegenden Wert am Operanden-Stack, weswegen dieser Wert im ersten Schritt bereits evaluiert wurde. Die konkrete Implementierung, wie in **??** zu sehen ist, wurde Dziworski [6] entnommen.

Quelltext 6.8: Bytecode zum Aufruf der `println` Funktion von Java

```
    private fun generate(printFunction: PrintFunction, scope: Scope) {
        val expression = printFunction.message
        mv.visitFieldInsn(Opcodes.GETSTATIC, "java/lang/System", "out", "Ljava/io/
        PrintStream;")
        expressionGenerator.generate(expression, scope)
        val type = expression.type
        val descriptor = "(\${type.getDescriptor()})V"
        val fieldDescriptor = "java/io/PrintStream"
        mv.visitMethodInsn(Opcodes.INVOKEVIRTUAL, fieldDescriptor, "println", descriptor
        , false)
    }
```

Kapitel 7

Tests

Um die Funktionalität von *toya* zu gewährleisten, sind Tests zu vollziehen. Diese Tests verlaufen in drei Schritte, wie folgt:

1. Programmcode in *toya* schreiben
2. Den kompilierten Bytecode analysieren
3. Die Ausgabe des Programms überprüfen

Die Analyse des Bytecode erfolgt mit dem, in der JDK inkludierten Werkzeug `javap`. Dieses erlaubt es, den Bytecode einer `class`-Datei in einer für Menschen leserlichen Form auszugeben. Der Aufruf erfolgt über die Kommandozeile im Format: `javap -c Main.class`. Das Argument `-c` zeigt zusätzlich die Bytecode Befehle innerhalb der Methoden an. Die Ausführung der Programme erfolgt über die Kommandozeile mit dem Befehl `java Main`.

Insgesamt gilt es, die einzelnen Sprachkonstrukte, wie zum Beispiel For-Schleifen und Variablendeklaration und anschließend umfangreichere Programme zu testen.

7.1 Hello World

Der erste Test stellt ein *Hello World* Programm dar, wie in Quelltext 7.1 zu sehen ist. Der Bytecode dieses Programms beschränkt sich auf einige wenige Befehle. Quelltext 7.2 zeigt den Bytecode. Konkret laden `getstatic` und `ldc` Referenzen zum `System.out` Objekt und der auszugebenden Zeichenkette. Anschließend ruft `invokevirtual` die `println` Funktion mit der Zeichenkette auf. Als Resultat gibt dieser Test die Zeichenkette *Hello World* auf der Konsole aus, zu sehen in Quelltext 7.3.

Quelltext 7.1: Quelltext des Hello World Programms

```
function main(args: string[]) {
    print("Hello World")
}
```

Quelltext 7.2: Bytecode des Hello World Programms

```
public class Main {
    public static void main(java.lang.String[]);
        Code:
            0: getstatic     #12     // Field java/lang/System.out:Ljava/io/PrintStream;
```

```
            3: ldc           #14     // String Hello World
            5: invokevirtual #20     // Method java/io/PrintStream.println:(Ljava/lang/
    String;)V
            8: return
}
```

Quelltext 7.3: Konsolen-Ausgabe des Hello World Programms

```
Hello World
```

7.2 Funktionen

Der zweite Test beschäftigt sich mit der Verwendung von Funktionen und dem Arbeiten mit Funktionsergebnissen. Quelltext 7.4 zeigt den Quelltext des Programms. Als erstes ruft dieser Test die `title` Funktion auf. Diese Funktion gibt die Zeichenkette *This is an addition:* auf der Konsole aus. Anschließend ruft das Programm die `print` Funktion mit dem Ergebnis der `add` Funktion auf. Die `add` Funktion addiert in diesem Fall die ganzzahligen Wertliterale `1` und `2`. Dementsprechend gibt die `print` Funktion den Wert `3` auf der Konsole aus, wie in Quelltext 7.6 zu sehen ist.

Der erste Test behandelt die Ausgabe via der `print` Funktion, daher wird an dieser Stelle nicht näher darauf eingegangen. Die `add` Funktion lädt mit den Opcodes `iload_0` und `iload_1` die beiden ganzzahligen Parameter der Funktion. `iadd` addiert anschließend die beiden Werte und liefert diese mit `ireturn` zurück. Der vollständige Bytecode für die drei Funktionen ist in Quelltext 7.5 zu sehen.

Quelltext 7.4: Funktionen

```
function title() {
    print("This is an addition:")
}

function add(lhs: int, rhs: int) -> int {
    lhs + rhs
}

function main(args: string[]) {
    title()
    print(add(1,2))
}
```

Quelltext 7.5: Bytecode für Funktionen

```
public class Main {
    public static void title();
        Code:
            0: getstatic     #12     // Field java/lang/System.out:Ljava/io/PrintStream;
            3: ldc           #14     // String This is an addition:
            5: invokevirtual #20     // Method java/io/PrintStream.println:(Ljava/lang/
    String;)V
            8: return

    public static int add(int, int);
        Code:
```

```
            0: iload_0
            1: iload_1
            2: iadd
            3: ireturn

    public static void main(java.lang.String[]);
        Code:
            0: invokestatic  #26    // Method title:()V
            3: getstatic     #12    // Field java/lang/System.out:Ljava/io/PrintStream;
            6: ldc           #27    // int 1
            8: ldc           #28    // int 2
            10: invokestatic  #30   // Method add:(II)I
            13: invokevirtual #33   // Method java/io/PrintStream.println:(I)V
            16: return
}
```

Quelltext 7.6: Konsolen-Ausgabe der Funktionen

```
This is an addition:
3
```

7.3 Variablen

Der dritte Test behandelt die Deklaration und Initialisierung von Variablen. Als erstes wird eine Variable pro Typ mit Wertliteralen und eine weitere Variable mit einem Funktionsergebnis angelegt, wie in Quelltext 7.7 zu sehen ist. Anschließend wird jeder Wert auf der Konsole ausgegeben, siehe dazu Quelltext 7.9.

Die beiden Opcodes `<type>load` und `<type>store` laden und speichern respektive den Wert einer lokalen Variable. Der Bytecode ist unter Quelltext 7.8 zu sehen.

Quelltext 7.7: Variablen

```
function someFunction() -> int {
    return 8
}

function main(args: string[]) {
    var number = 123
    var word = "Hello World"
    var double = 123.456
    var bool = true
    var result = someFunction()

    print(number)
    print(word)
    print(double)
    print(bool)
    print(result)
}
```

Quelltext 7.8: Bytecode von Variablen

```
public class Main {
    public static int someFunction();
        Code:
```

```
            0: ldc           #7                  // int 8
            2: ireturn

    public static void main(java.lang.String[]);
        Code:
            0: ldc           #10    // int 123
            2: istore_1
            3: ldc           #12    // String Hello World
            5: astore_2
            6: ldc2_w        #13    // double 123.456d
            9: dstore_3
            10: iconst_1
            11: istore        5
            13: invokestatic  #16   // Method someFunction:()I
            16: istore        6
            18: getstatic     #22   // Field java/lang/System.out:Ljava/io/PrintStream;
            21: iload_1
            22: invokevirtual #28   // Method java/io/PrintStream.println:(I)V
            25: getstatic     #22   // Field java/lang/System.out:Ljava/io/PrintStream;
            28: aload_2
            29: invokevirtual #31   // Method java/io/PrintStream.println:(Ljava/lang/
    String;)V
            32: getstatic     #22   // Field java/lang/System.out:Ljava/io/PrintStream;
            35: dload_3
            36: invokevirtual #34   // Method java/io/PrintStream.println:(D)V
            39: getstatic     #22   // Field java/lang/System.out:Ljava/io/PrintStream;
            42: iload         5
            44: invokevirtual #37   // Method java/io/PrintStream.println:(Z)V
            47: getstatic     #22   // Field java/lang/System.out:Ljava/io/PrintStream;
            50: iload         6
            52: invokevirtual #28   // Method java/io/PrintStream.println:(I)V
            55: return
}
```

Quelltext 7.9: Konsolen-Ausgabe der Variablen

```
123
Hello World
123.456
true
8
```

7.4 Felder

Der vierte Test behandelt die Deklaration, Initialisierung und den Zugriff auf Felder. Der Quelltext ist in Quelltext 7.10 zu sehen. Das Programm legt an erster Stelle ein acht Elemente großes Feld vom Typ `int` an. Dem Index `3` wird das Wertliteral `15` zugewiesen. Anschließend gibt das Programm die Indizes `3` und `4` auf der Konsole aus. Wie erwartet liegt in `arr[3]` der Wert `15` und in `arr[4]` der Wert `0`, wie an der Konsolenausgabe in Quelltext 7.12 zu sehen ist. Primitive Werte haben in Java immer einen Standardwert, was bei Ganzzahlen `0` ist. Obwohl `arr[4]` kein Wert zugewiesen wurde, ist daher trotzdem `0` auf der Konsole zu sehen.

Der Opcode `newarray` legt ein neues Feld an. Mit den Opcodes `iaload` und `iastore`

ist der Zugriff auf Elemente des Feldes möglich. Der Bytecode ist in Quelltext 7.11 zu sehen.

Quelltext 7.10: Felder

```
function main(args: string[]) {
    var arr = new int[8]
    arr[3] = 15

    print(arr[3])
    print(arr[4])
}
```

Quelltext 7.11: Bytecode von Felder

```
public class Main {
    public static void main(java.lang.String[]);
        Code:
             0: ldc           #7    // int 8
             2: newarray       int
             4: astore_1
             5: aload_1
             6: ldc           #8    // int 3
             8: ldc           #9    // int 15
            10: iastore
            11: getstatic     #15   // Field java/lang/System.out:Ljava/io/PrintStream;
            14: aload_1
            15: ldc           #8    // int 3
            17: iaload
            18: invokevirtual #21   // Method java/io/PrintStream.println:(I)V
            21: getstatic     #15   // Field java/lang/System.out:Ljava/io/PrintStream;
            24: aload_1
            25: ldc           #22   // int 4
            27: iaload
            28: invokevirtual #21   // Method java/io/PrintStream.println:(I)V
            31: return
}
```

Quelltext 7.12: Konsolen-Ausgabe der Felder

```
15
0
```

7.5 If-Verzweigungen

Der fünfte Test behandelt die Verwendung von If-Verzweigungen als Ausdruck und Anweisung. Der erste Teil des Quelltextes verwendet die If-Verzweigung als Ausdruck um einer Variable einen Wert zuzuweisen und diese auf der Konsole auszugeben. Der zweite Teil verwendet die If-Verzweigung als Anweisung um eine Zeichenkette je nach Zweig auszugeben. Der Quelltext ist in Quelltext 7.13 zu sehen. Quelltext 7.15 zeigt die Konsolenausgabe.

Im Bytecode sind die `goto` Opcodes zu beachten. Mithilfe `goto` ist der Sprung zwischen Anweisungen im Bytecode möglich. Jede Anweisung ist mit einer Nummer verse-

hen. Diese Nummer ist bei `goto` anzugeben, um den Zielort des Sprungs zu bestimmen. Der Bytecode ist unter Quelltext 7.14 zu finden.

Quelltext 7.13: If-Verzweigungen

```
function main(args: string[]) {
    var value = if (3 > 4) 5 else 6
    print(value)

    if(3 < 4) {
        print("true branch")
    } else {
        print("false branch")
    }
}
```

Quelltext 7.14: Bytecode der If-Verzweigungen

```
public class Main {
    public static void main(java.lang.String[]);
        Code:
             0: ldc           #7    // int 3
             2: ldc           #8    // int 4
             4: if_icmpgt     11
             7: iconst_0
             8: goto          12
            11: iconst_1
            12: ifne          20
            15: ldc           #9    // int 6
            17: goto          22
            20: ldc           #10   // int 5
            22: istore_1
            23: getstatic     #16   // Field java/lang/System.out:Ljava/io/PrintStream;
            26: iload_1
            27: invokevirtual #22   // Method java/io/PrintStream.println:(I)V
            30: ldc           #7    // int 3
            32: ldc           #8    // int 4
            34: if_icmplt     41
            37: iconst_0
            38: goto          42
            41: iconst_1
            42: ifne          56
            45: getstatic     #16   // Field java/lang/System.out:Ljava/io/PrintStream;
            48: ldc           #24   // String false branch
            50: invokevirtual #27   // Method java/io/PrintStream.println:(Ljava/lang/
    String;)V
            53: goto          64
            56: getstatic     #16   // Field java/lang/System.out:Ljava/io/PrintStream;
            59: ldc           #29   // String true branch
            61: invokevirtual #27   // Method java/io/PrintStream.println:(Ljava/lang/
    String;)V
            64: return
}
```

Quelltext 7.15: Konsolen-Ausgabe der If-Verzweigungen

```
6
true branch
```

7.6 For-Schleifen

Der sechste Test behandelt For-Schleifen. Der Schleifenkopf initialisert und deklariert eine Zählvariable `i` mit dem Wert `0`. Nach jedem Schleifendurchlauf prüft die Abbruchbedingung, ob `i` den Wert der zuvor angelegten Variable `n` überschreitet. `n` wurde mit dem Wert `200` initialisert. Der Wert von `i` erhöht sich nach jedem Schleifendurchlauf um zehn. Im Schleifenkörper befindet sich eine `print`-Anweisung, die den Wert der Zählvariable auf der Konsole ausgibt. Der Quelltext ist in Quelltext 7.16 zu sehen. Neben If-Verzweigungen sind `goto` Anweisungen auch bei For-Schleifen in Verwendung. Nach jedem Schleifendurchlauf springt das Programm zur Abbruchbedingung zurück. Evaluiert die Abbruchbedingung zu `true`, springt das Programm zum `return` und beendet die Ausführung. Der Bytecode ist unter Quelltext 7.17 zu sehen. Auf der Konsole sind die Werte von 0 bis 200 in zehner Inkrementen zu sehen. Die Ausgabe ist in Quelltext 7.18 zu finden.

Quelltext 7.16: For-Schleifen

```
function main(args: string[]) {
    var n = 200

    for (var i = 0; i <= n; i = i+10) {
        print(i)
    }
}
```

Quelltext 7.17: Bytecode der For-Schleife

```
public class Main {
    public static void main(java.lang.String[]);
        Code:
             0: ldc           #7    // int 200
             2: istore_1
             3: ldc           #8    // int 0
             5: istore_2
             6: iload_2
             7: iload_1
             8: if_icmple     15
            11: iconst_0
            12: goto          16
            15: iconst_1
            16: ifeq          34
            19: getstatic     #14   // Field java/lang/System.out:Ljava/io/PrintStream;
            22: iload_2
            23: invokevirtual #20   // Method java/io/PrintStream.println:(I)V
            26: iload_2
            27: ldc           #21   // int 10
            29: iadd
            30: istore_2
            31: goto          6
            34: return
}
```

Quelltext 7.18: Konsolen-Ausgabe der For-Schleife

```
0
```

```
10
20
30
40
50
60
70
80
90
100
110
120
130
140
150
160
170
180
190
200
```

7.7 Umfangreicheres Beispiel

Die Tests bisher beschränkten sich auf einzelne Eigenschaften von *toya*. Um aber auch das Zusammenspiel von mehreren Eigenschaften zu testen, wird nun noch ein umfangreicheres Programm getestet. Dieses Programm ruft eine Funktion auf, welche Zeichenketten auf der Konsole ausgibt, definiert Variablen von Felder und primitiven Datentypen, iteriert über eine For-Schleife, welche eine If-Verzweigung enthält und verändert und liest Felder. Der Quelltext ist in Quelltext 7.19 zu sehen.

Quelltext 7.19: Quelltext des umfangereicheren Beispiels

```
function printIntro(n: int) {
    print("Welcome to a simple toya program")
    print("Calculating numbers up to: ")
    print(n)
    print("---------")
}

function main(args: string[]) {
    var n = 200
    printIntro(n)

    for (var i = 0; i <= n; i = i+10) {
        if (i == 20) {
            print("i is 20:")
        }
        print(i)
    }

    var boolArr = new boolean[2]
    boolArr[1] = true
    print(boolArr[0])
    print(boolArr[1])
}
```

Da das Anzeigen des Bytecodes bei diesem umfangereicheren Beispiel zu lange ist und alle Fragmente in den anderen Tests bereits zu sehen sind, wird der Bytecode an dieser Stelle ausgelassen. Quelltext 7.20 zeigt die Konsolenausgabe des siebten Tests.

Quelltext 7.20: Konsolen-Ausgabe des umfangereicheren Beispiels

```
Welcome to a simple toya program
Calculating numbers up to:
200
---------
0
10
i is 20:
20
30
40
50
60
70
80
90
100
110
120
130
140
150
160
170
180
190
200
false
true
```

Kapitel 8

Zusammenfassung, Schlüsse und Lehren

Ziel dieser Bachelorarbeit war es, sich mit der Implementierung einer eigens konzipierten Programmiersprache auseinanderzusetzen. Dieser Aufgabenstellung konnte erfolgreich durch die Implementierung von *toya* und dessen Compiler nachgegangen werden. *Toya* bietet in finaler Form die Definition von Funktionen, Variablen, Kontrollflüssen und arithmetischen Ausdrücken. *Toya* ist statisch typisiert und bietet als Datentypen Ganz- und Gleitkommazahlen, Zeichenketten und boole'sche Werte zur Initialisierung von Variablen, Funktionsparametern und Rückgabewerten.

Positiv hervorzuheben ist die Arbeit mit ANTLR. Nicht aufgrund der Effizienz, sondern aufgrund der ausgezeichneten Eingliederung in den Arbeitsablauf stellt es sich als besonders gutes Entwicklungswerkzeug dar. Mithilfe eines Intellij IDEA Plugins können Textfragmente hinsichtlich Ihrer Validität für eine spezifizierte Grammatik verifiziert und mögliche Fehler umgehend erkannt und behoben werden. Die Grammatikdefinition anhand g4-Dateien ermöglicht eine klare Übersicht darüber, wie die Programmiersprache nun konkret strukturiert ist.

Toya und dessen Komponenten können auch leicht getestet werden. Als Eingabewert dienen Sprachfragmente, wie zum Beispiel eine Variablendeklaration. Als Ausgabewert, welcher anschließend auf Validität des Ergebnisses zu vergleichen ist, kann der generierte Bytecode verwendet werden. Durch diese textuelle Ein- und Ausgabe wird *toya* zur leicht testbaren Sprache. Wächst nun der Umfang von *toya*, so steigt der Testaufwand nicht überproportional, sondern beschränkt sich auf die neuen Sprachaspekte.

Sollte Interesse an der Weiterentwicklung von *toya* bestehen, wäre es sinnvoll, ANTLR durch einen eigens implementierten Analysator zu ersetzen, da der Zeitaufwand des vollständigen Syntaxbaums mit steigender Komplexität erheblich zunimmt. Ein eigens implementierter Analysator hingegen kann viele Teile des erzeugten Syntaxbaums völlig verwerfen, da für gewöhnlich nur ein Teil aller Informationen für die schlussendliche Erzeugung des Maschinencodes relevant ist. Für weniger umfangreiche Grammatiken erweist sich ANTLR als sinnvoll und ist daher ohne Bedenken weiterzuempfehlen.

Quellenverzeichnis

[1] Eric Bruneton, Romain Lenglet und Thierry Coupaye. „ASM: a code manipulation tool to implement adaptable systems“. *Adaptable and extensible component systems* 30.19 (2002) (siehe S. 25).

[2] Tim Lindholm u. a. *The Java virtual machine specification: Java SE 8 edition.* 2016 (siehe S. 16, 19).

[3] Robert C Martin. *Clean code: a handbook of agile software craftsmanship.* Pearson Education, 2009 (siehe S. 5).

[4] Oracle. *HotSpot Runtime Overview.* URL: https://openjdk.org/groups/hotspot/docs/RuntimeOverview.html (besucht am 20. 05. 2023) (siehe S. 16).

[5] Terence Parr, Sam Harwell und Kathleen Fisher. „Adaptive LL (*) parsing: the power of dynamic analysis“. *ACM SIGPLAN Notices* 49.10 (2014), S. 579–598 (siehe S. 13).

[6] Jakub Dziworski. *Creating JVM language - Enkel.* URL: http://jakubdziworski.github.io/enkel/2016/03/10/enkel_first.html (besucht am 18. 07. 2022) (siehe S. 28).

[7] Google. *Android's Kotlin-first approach.* 2019. URL: https://developer.android.com/kotlin/first (besucht am 10. 05. 2023) (siehe S. 9).

[8] JetBrains. *Kotlin Documentation.* URL: https://kotlinlang.org/docs/home.html (besucht am 10. 05. 2023) (siehe S. 9).

[9] Terence Parr. *About The ANTLR Parser Generator.* URL: https://www.antlr.org/about.html (besucht am 25. 02. 2023) (siehe S. 13).

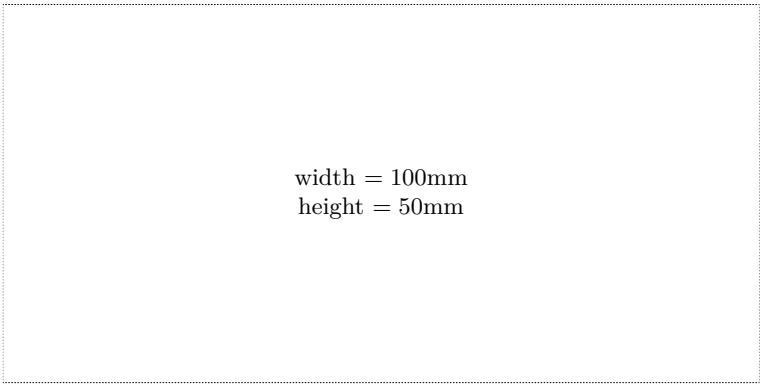
[10] Terence Parr. *ANTLR.* 25. Feb. 2023. URL: https://www.antlr.org/ (besucht am 25. 02. 2023) (siehe S. 13).

[11] Terence Parr u. a. *Runtime Library and Code Generation Targets.* 2022. URL: https://github.com/antlr/antlr4/blob/master/doc/targets.md (besucht am 20. 05. 2023) (siehe S. 13).

[12] Gabriele Tomassetti. *The ANTLR Mega Tutorial.* 2021. URL: https://tomassetti.me/antlr-mega-tutorial/ (besucht am 18. 01. 2023) (siehe S. 13).

Messbox zur Druckkontrolle

— Druckgröße kontrollieren! —



— Diese Seite nach dem Druck entfernen! —