toya: Eine imperative Programmiersprache für die JVM

Lukas Christian Hofwimmer

MASTERARBEIT

eingereicht am
Fachhochschul-Bachelorstudiengang

Universal Computing

in Hagenberg

im Juli 2021

Betreuung:

FH-Prof. DI Dr. Heinz Dobler

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 Arbeit ist mit dem elektronisch übermittelten Textdokument identisch.

Hagenberg, am 15. Juli 2021

Lukas Christian Hofwimmer

Inhaltsverzeichnis

Vorwort

Kurzfassung

Abstract

This should be a 1-page (maximum) summary of your work in English.

Kapitel 1

Einleitung

Kapitel 2

Die Programmiersprache toya

Toya ist eine stark typisierte, turing-vollständige Programmiersprache für die Java Virtual Machine mit einem Fokus auf Simplizität. Der Syntax ist, wie C# oder Java zum Beispiel, stark an C angelehnt. Auf den Syntax wird in diesem Kapitel bei der näheren Behandlung der einzelnen Komponenten der Sprache eingegangen.

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

Grundsätzlich folgt toya imperativen Programmierparadigmen. Ein toya-Programm besteht aus einer Menge an Funktionen und Variablen, wobei, wie in Java eine `main` Funktion zum Programmeinstieg benötigt wird. Funktionen beinhalten verschiedene Klassen gibt es keine; bei der Kompilation werden aber alle Programmteile in eine `Main` Klasse zusammengefasst, da jede *.class* Datei genau eine Klasse beinhalten muss. Sollte die `main` Funktion nicht vorhanden sein, so wird eine Exception während des Parsens geworfen. Variablen, die außerhalb von Funktionen definiert werden, können global verwendet werden. Global in diesem Kontext bedeutet, dass Variablen in allen Funktionen des Programms verwendet werden können.

2.1 Typen

Toya stellt insgesamt 5 Typen und deren Array-Gegenstücke zur Verfügung. Diese sind `boolean`, `int`, `double` und `string`, wobei hierbei `int`, `boolean` und `string` eindeutig die wichtigsten sind, da jeder dieser Typen in unterschiedlichen Domänen seine Verwendung findet. So werden boolean Das erstellen weiterer Typen ist nicht möglich.

Array-Typen werden über den allgemein bekannten Suffix '`[]`' deklariert. So ist zum Beispiel der Typ eines String-Arrays als `string[]` zu schreiben. Die JVM bietet zusätzlich noch die Datentypen `byte`, `short`, `long` und `float` an, aber weil alle dieser Typen redundant in ihrem Verwendungszweck sind, kommen diese nicht in *toya* vor, da der Sinn von *toya* nicht die vollständige Ausschöpfung aller JVM-Features ist, sondern die explorative Implementierung einer Programmiersprache, wofür nicht alle Datentypen benötigt werden. Der `returnAddress` Typ wird hier außer acht gelassen, weil dieser nur JVM-interne Relevanz hat.

Integer besitzt 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 kann die Werte `true` und `false` annehmen. `True` wird intern als 1 gehandhabt, `false` als 0. Die JVM besitzt keinen nativen String Typen, da dieser als Referenzwert gehandhabt wird. Da *toya* keine Erstellung von Typen erlaubt, sind die einzigen Referenztypen String und Arrays. Die Bytecode-Generierung *toya's* unterscheidet immer zwischen Arrays und nicht-Arrays, wenn es um die Auswahl der richtigen Opcodes geht, dadurch ergibt sich, dass eine Referenz, welche kein Array ist, immer ein String sein muss. Strings werden als Literale via doppelte Anführungszeichen definiert. So ist ein Hello World String als `"Hello World"` anzugeben.

Gewisse Sprachen, wie Java oder Kotlin konvertieren automatisch Typen, wenn diese zum Beispiel in arithmetischen Operationen gemischt werden. So wird zum Beispiel eine Addition, die aus einem Integer- und Double-Wert besteht, automatisch zum Typen Double konvertiert. Dadurch ermöglichen diese Programmiersprachen eine gewisse Flexibilität, selbst bei statischer Typisierung. Diese automatische Konvertierung besitzt *toya* nicht, stattdessen wirft der Compiler einen Fehler, sollten verschiedene Typen in einer Operation vorkommen.

2.2 Funktionen

Funktionen sind die zentrale Komponente von *toya* und beinhalten die Programmlogik. Sie bestehen aus einem Funktionskörper und der Funktionssignatur. Die Funktionssignatur beinhaltet Name der Funktion, Parameter und Rückgabewert. Der Name ist das einzig verpflichtende hierbei; Parameter und Rückgabewert sind rein optional. Hat eine Funktion keinen Rückgabewert, so ist in der Funktion der Pfeil, als auch der nachfolgende Typ wegzulassen.

Listing 2.1: Eine typische Funktion unter toya.

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

Der Funktionskörper beinhaltet eine beliebige Menge an Statements und Ausdrücken. Hat eine Funktion einen Rückgabewert, so kann mit `return` ein nachfolgender Ausdruck rückgegeben werden. Das Schlüsselwort `return` ist jedoch optional: Wenn das letzte Statement gleichzeitig ein Ausdruck und vom gefordertem Typen ist, dann wird automatisch dieser Ausdruck retourniert. Hierbei ist jedoch aufzupassen, dass die Leserlichkeit erhalten bleibt.

Funktionen werden mit dem Syntax `<funcname>(parameter*)` aufgerufen und sind vom Rückgabetypen der aufgerufenen Funktion. Für jeden Parameter kann jeder beliebige Ausdruck eingesetzt werden, solange der Ist- und Solltyp übereinstimmt.

2.3 Statements

Statements sind Programmanweisungen, die keinen Wert zurückgeben. Dazu gehören Variablendeklaration und -zuweisung, For-Schleifen und Return-Anweisungen.

2.3.1 Variablen

Toya erlaubt der Nutzerin die Erstellung von Variablen in Funktionen und auf globaler Ebene. Der Syntax dafür lautet `var <name> = <ausdruck>`; die explizite Angabe eines Typens ist nicht möglich. Stattdessen inferiert der Compiler anhand bekannter Typinformation des zu evaluierenden Ausdrucks den Typen und weist diesen Typen der Variable zu. Dieser Typ bleibt über die gesamte Lebensdauer der Variable gleich. Sobald der Typ einer Variable einmal fixiert ist, so kann dieser Typ nicht mehr geändert werden. Initialisiert man also eine Variable mithilfe eines Ausdrucks, der zu `int` evaluiert, so ist die Variable bis zur Vernichtung durch den Garbage Collector vom Typen `int`. Eine getrennte Deklaration und Initialisierung ist nicht möglich.

Abgesehen von typentheoretischer Relevanz bietet die Verwendung des Schlüsselwortes `var` einige Vorteile als auch Nachteile für Verwenderinnen von *toya*. Da `var` eine Vielzahl von verschiedenen Typen ersetzt, erleichtert es die Schreibarbeit für Programmiererinnen ungemein. Robert C. Martin sagt jedoch in seinem nominalen Werk *A handbook of agile software craftmanship* "Code is more read than it is written.". Daraus folgt, dass die Lesbarkeit wichtiger als *Schreibbarkeit* von Code ist und hierbei zeigen sich dann auch die Schwächen.

Ohne einer modernen Entwicklungsumgebung, welche via dem User-Interfaces Hinweise auf die Typisierung gibt, kann die Entwicklerin nur über den konkreten Typen Vermutungen anstellen. Aufgrund der geringen Anzahl an Typen in *toya* ist das Fehlen von Typhinweisen jedoch vernachlässigbar. Vergleicht man nun die Variablendeklaration und Initialisierung mit Java, so ist zu erkennen, dass bei der Initialisierung via Literalen die Verwendung von `var` kein Problem darstellt. Will man einer Variable den retournierten Wert einer Funktion zuweisen, so können hier aber Schwierigkeiten hinsichtlich Schlussfolgerungen auftreten. Daher ist die bewusste und intelligente Vergabe von Variablennamen essenziell.

Listing 2.2: Variablendeklaration in toya

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

Listing 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 eigenen Scope oder Elternscope verwendet werden, da es ansonsten zu Unklarheiten kommen kann, welche von mehreren Variablen nun gemeint ist. Auf die Frage, was sich hinter einem Scope verbiergt, wird näher im Kapitel 7 eingegangen. Ein Name darf aus beliebig vielen Groß- und Kleinbuchstaben und Unterstrichen bestehen.

2.3.2 Arrays

Arrays sind eine eindimensionale Liste von einem bestimmten Typen mit einer fixen Länge, welche bei der Deklaration des Arrays angegeben wird und sich über die Lebensdauer der Variable nicht mehr ändert. Zuweisungen und Deklarationen unterscheiden sich nur leicht von nicht-Array Variablen. Der Hauptunterschied dabei besteht darin, dass die Länge bei der Deklaration und der Index bei der Zuweisung angegeben werden muss.

Arrays werden mit dem Syntax `var <name> = new <typ>[int-Ausdruck]` deklariert. Mit `<name>[int-Ausdruck] = Ausdruck` wird ein neuer Wert auf einen Index geschrieben.

2.3.3 For-Schleifen

For-Schleifen in *toya* verhalten sich gleich wie in vielen anderen Sprachen, Java zum Beispiel. Sie bestehen aus einem Schleifenkopf und einem Schleifenkörper. Der Schleifenkopf besteht aus 3 Teilen von denen alle 3 optional sind:

- **Zählvariable:** Eine Variablendeklaration, die üblicherweise vom Inkrement-Ausdruck verändert wird und womit die Abbruchbedingung arbeitet.
- **Abbruchbedingung:** Ein Ausdruck, der zu einem booleschen Wert evaluieren muss.
- **Inkrement-Ausdruck:** Der Ausdruck, welcher nach jedem Schleifendurchlauf aufgerufen wird und wo typischerweise die Zählvariable verändert wird.

Listing 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 man also mit `for (;;) ...` eine Endlos-Schleife erzeugen.

2.4 Ausdrücke

Ausdrücke in *toya* sind alle Konstrukte, welche einen Wert zurückgeben. So sind Funktionsaufrufe, die einen Rückgabewert haben, If-Verzweigungen, boolesche und arithmetische Ausdrücke, Literale und Variablen. Ausdrücke evaluieren immer zu einem konkreten Wert, welcher aus dem Wertebereich eines bestimmten Typen entstammt. Abgesehen vom zwingenden Übereinstimmen des Ist- und Solltypen existieren keine Beschränkungen, was das Ersetzen von Ausdrücke durch andere Ausdrücken betrifft.

2.4.1 Arithmetik

Toya unterstützt Multiplikation (`*`), Addition (`+`), Subtraktion (`-`) und Division (`/`) für Integer- und Double-Werte via Infix Notation. Bei der Evaluierung von arithmetischen Ausdrücke wird auf die Klammerung und auf die Operatorrangfolge (*Punkt vor Strich*) Rücksicht genommen, da die Evaluierung immer von innen nach außen geschieht.

2.4.2 Boolesche Logik

Boolesche Ausdrücke sind essenziell für Verwendung von If-Verzweigungen und For-Schleifen, da sie als Abbruchbedingung, beziehungsweise für die Zweig-Wahl benötigt werden. Zur Auswahl stehen die booleschen Operatoren 'Und' (`&&`) und 'Oder' (`||`) und die Comparators Größer (`>`), Größer-Gleich (`>=`), Gleich (`==`), Kleiner-Gleich (`<=`) und Kleiner (`<`). Außerdem können boolesche Ausdrücke mit dem Präfix `!` negiert werden.

Sollten Teile eines booleschen Ausdrucks geklammert sein, so nimmt der Parser Rücksicht darauf, jedoch sind alle Operatoren gleichrangig.

2.4.3 If-Verzweigungen

If-Verzweigungen ermöglichen der Entwicklerin, bedingte Ausführung zu implementieren. If-Verzweigungen sind sehr flexibel in ihrem Syntax, da sowohl ein einzelner Ausdruck als auch mehrere Statements in einem Zweig vorkommen können. `if (<boolean-ausdruck>) { <statement>* } else { <statements>* }` ermöglicht mehrere Statements pro Block, wobei jeder Block von geschwungenen Klammern umgeben ist. Ist das letzte Statement ein Ausdruck, so oder `if (<boolean-ausdruck>) ausdruck else ausdruck`

Listing 2.5: If-Verzweigung als klassisches Statement.

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

Listing 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 Statement 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.

Um die Verwendung als Ausdrück zu ermöglichen, ist zu beachten, dass bei If-Verzweigungen im Statement-Format das letzte Statement ein Ausdruck sein muss; beim Ausdruck-Format besteht jeder Zweig sowieso nur aus einem Ausdruck, dadurch ist diese Bedingung hinfällig. Außerdem müssen die Typen in allen Zweigen übereinstimmen und der ansonsten optionale else-Zweig zwingend vorhanden sein. Wäre der else-Zweig nämlich nicht vorhanden, dann wäre kein Wert als Folge der Evaluierung garantierbar.

2.4.4 Literale

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

- **Integer:** Alle numerischen Werte ohne Nachkommastellen. (z.B.: `12345`)
- **Double:** Alle numerischen Werte mit Nachkommastellen. (z.B.: `123.45`)

- **String:** Alle Zeichenketten, die von doppelten Anführungszeichen umgeben sind. (z.B.: `"Hello World"`)
- **Boolean:** `true` und `false`.

2.5 Kommentare

Toya erlaubt die Erstellung von Kommentaren. Ein Kommentar beginnt mit einem doppelten Schrägstrich `//`; alle weiteren Zeichen in dieser Zeile – bis zum Zeilenumbruch – ignoriert der Parser.

Kapitel 3

Vergleich mit Kotlin

Kapitel 4

Generierung des Syntaxbaums mit ANTLR

Kapitel 5

Die JVM und deren Bytecode

Kapitel 6

Generierung des Bytecodes mit ASM

Kapitel 7

Implementierung von toya

Lorem ipsum dolor sit amet.

Kapitel 8

Tests von toya

Kapitel 9

Fazit, Schlüsse und Lehren

Anhang A

Technische Informationen

Anhang B

Ergänzende Inhalte

Auflistung der ergänzenden Materialien zu dieser Arbeit, die zur digitalen Archivierung an der Hochschule eingereicht wurden (als ZIP-Datei).

B.1 PDF-Dateien

Pfad: /

thesis.pdf . . . . . Finale Master-/Bachelorarbeit (Gesamtdokument)

B.2 Mediendaten

Pfad: /media

*.ai, *.pdf . . . . . Adobe Illustrator-Dateien
*.jpg, *.png . . . . . Rasterbilder
*.mp3 . . . . . Audio-Dateien
*.mp4 . . . . . Video-Dateien

B.3 Online-Quellen (PDF-Kopien)

Pfad: /online-sources

Reliquienschrein-Wikipedia.pdf

Anhang C

Fragebogen

Anhang D

LaTeX-Quellcode

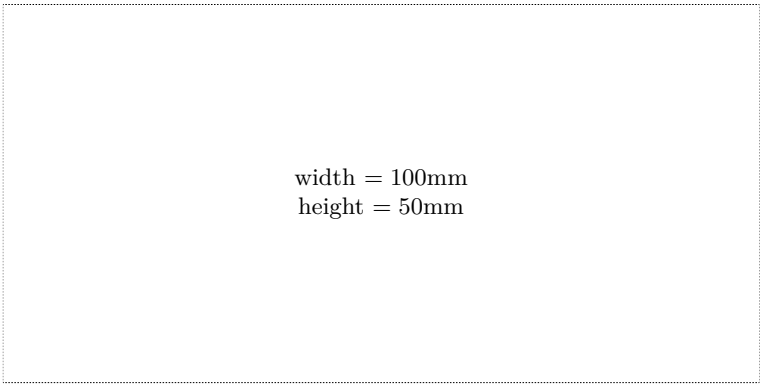
Quellenverzeichnis

Online-Quellen

[1] *Reliquienschrein*. 20. Okt. 2020. URL: https://de.wikipedia.org/wiki/Reliquienschrein (besucht am 12. 05. 2021).

Messbox zur Druckkontrolle

— Druckgröße kontrollieren! —



— Diese Seite nach dem Druck entfernen! —