# Inhaltsverzeichnis


**1 Einleitung** **1**

**2 Das Plugin Java2Jinja** **2**
2.1 Vorentscheidungen . . . 2
2.2 Systemvoraussetzungen und Installation . . . 3
2.3 Benutzeroberfläche . . . 4
2.4 Standardbibliothek . . . 6

**3 Jinja** **6**
3.1 Typen und Werte . . . 7
3.2 Ausdrücke . . . 8
3.3 Programme . . . 9
3.4 Wohlgeformtheitsbedingungen . . . 9

**4 Unterstützte Sprachkonstrukte** **10**
4.1 Notation . . . 10
4.2 Überblick . . . 11
4.3 Typen . . . 11
4.3.1 char . . . 11
4.3.2 String . . . 11
4.4 Deklarationen . . . 13
4.4.1 Klassendeklarationen . . . 13
4.4.2 Felddeklarationen . . . 13
4.4.3 Methodendeklarationen . . . 14
4.4.4 Statische Felder und Methoden . . . 16
4.5 Blöcke und lokale Variablen . . . 19
4.6 Rücksprung . . . 20
4.7 Bedingungen . . . 21
4.7.1 if . . . 21
4.7.2 switch . . . 22
4.8 Schleifen . . . 23
4.8.1 for . . . 23
4.8.2 do . . . 24
4.8.3 break und continue . . . 24
4.9 Klasseninstantiierung . . . 26
4.10 Array-Erzeugung . . . 27
4.11 Operatoren . . . 28
4.11.1 Unäre Operatoren . . . 28
4.11.2 Zuweisungsoperatoren . . . 30
4.11.3 Binäre Operatoren . . . 31
4.11.4 Stringkonkatenation . . . 31
4.12 Ausnahmebehandlung . . . 32

4.13 Zusicherungen . . . . . 34

**5 Abhängigkeitsauflösung** **34**

**6 Beispielprogramm** **36**

**7 Ausblick** **39**
7.1 Bisher nicht unterstützte Sprachelemente . . . . . 39
7.2 Vorschläge für Jinja-Erweiterungen . . . . . 40

**Literaturverzeichnis** **iii**

**Eidesstattliche Erklärung** **iv**

# 1 Einleitung

Eine formale Semantik erlaubt es, weitreichende Erkenntnisse über die semantischen Eigenschaften eines Programms durch mathematisch beweisbar korrekte Methoden zu erzielen. Damit wird es möglich, die Korrektheit oder andere Aspekte einer vorliegenden Implementierung zu beweisen, indem entsprechende Analysen auf der Semantik ausgeführt und die Ergebnisse auf den ursprünglichen Code übertragen werden. Mithilfe solcher Analysen lässt sich die Typkorrektheit eines Programms auf formal korrekte Weise ermitteln, was bei der klassischen Typprüfung in einem Übersetzer nicht gegeben ist. Es lassen sich sogar noch weitere Teile eines Übersetzers umsetzen und so aus der Semantik am Ende wieder ausführbarer Code erzeugen.

Mit Jinja, beschrieben von Klein und Nipkow [KN06], existiert seit einiger Zeit eine von Java abgeleitete Sprache, für die eine solche formale Semantik zur Verfügung steht. Jedoch ist diese Sprache gegenüber Java im Umfang reduziert und in bestimmten Details verändert, weshalb die händische Umformung von bestehendem Java-Code in die Sprache Jinja aufwändig und daher bisher wenig Jinja-Code vorhanden ist.

Diese Problematik soll mithilfe dieser Studienarbeit reduziert werden, indem ein maschineller Übersetzer entwickelt wird, der einen Teil des Sprachumfangs von Java in Jinja umsetzen kann. Dabei kommt es besonders auf die semantische Korrektheit der Übersetzung an, da hiervon direkt die Ergebnisse der Analysemethoden abhängen und eine fehlerhafte Übersetzung somit zu nicht verwertbaren Ergebnissen führt. Aus diesem Grund ist auch die Einfachheit des Vorgangs eine wichtige Zielsetzung, da eine hohe Komplexität Fehler in der Implementierung begünstigt, deren Existenz sich mangels einer formalen Semantik für Java nicht durch eine Verifikation der Implementierung verneinen lässt. Diese Ziele wurden weitgehend erreicht, indem ein Eclipse-Plugin mit dem Namen „Java2Jinja“ entwickelt wurde, welches die Verarbeitung von Java-Code innerhalb von Eclipse-Projekten unterstützt. Die Menge der unterstützten Sprachelemente genügt, um wesentliche Teile einer unabhängigen Implementierung der Java-Standardbibliothek unverändert verwenden zu können und damit sinnvolle Programme zu implementieren. Mithilfe einer Wohlgeformtheitsprüfung für die Sprache Jinja wurde überprüft, dass die erzeugten Programme zumindest korrekte Jinja-Programme sind.

In den weiteren Kapiteln dieser Studienarbeit wird erläutert, welche Ansätze bei der Implementierung gewählt wurden, wie das Plugin benutzt wird und was es leisten kann. In Kapitel 2 sind technische Aspekte der Entwicklung dargestellt und es wird die Verwendung durch den Benutzer erklärt. Kapitel 3 gibt einen kurzen Überblick über die Sprache Jinja, um die Voraussetzungen zu schaffen, die Ausführungen im folgenden Kapitel nachzuvollziehen. Der Hauptteil der Studienarbeit ist in den Kapiteln 4 und 5 zu finden, wo das Vorgehen bei der Übersetzung der unterstützten Sprachelemente angedeutet wird. Kapitel 6 gibt mithilfe eines Beispielprogramms einen Eindruck von den Fähigkeiten des Plugins und schließlich wird in Kapitel 7 auf zukünftig mögliche Erweiterungen dieser Fähigkeiten eingegangen.

# 2 Das Plugin Java2Jinja

## 2.1 Vorentscheidungen

Die Vorgaben für die Implementierung bestanden darin, dass als Eingabe Quellcodedateien der Sprache Java akzeptiert werden müssen und die Ausgabe eine Darstellung des abstrakten Syntaxbaums von Jinja in einer Programmiersprache sein muss, die auch Isabelle/HOL erzeugen kann, also ML, OCaml oder Haskell. Letztere Voraussetzung ist dadurch bedingt, dass die Ausgabe als Eingabe für weitere Schritte dienen können soll, Wohlgeformtheitsprüfung und Bytecodeerzeugung, die mithilfe von Isabelle/HOL implementiert und in eine dieser Sprachen exportiert werden. Diese exportierten Programme werden dann gemeinsam in einer Umgebung mit der Ausgabe von Java2Jinja ausgeführt und können so die Strukturen verarbeiten.

Die Arbeit der zu implementierenden Anwendung ließ sich absehbar in drei wesentliche Schritte zerlegen:

1. Zunächst ist die Erzeugung eines abstrakten Syntaxbaumes(AST) aus dem Quellcode der Eingabedatei notwendig, um die dahinterstehende Struktur algorithmisch verarbeiten zu können.

2. Es folgt eine Umwandlung in einen AST, der Jinja-Ausdrücke darstellen kann.

3. Zuletzt wird der Jinja-AST wieder in Textform umgewandelt, so dass er sich als Quellcodedatei einer der möglichen Programmiersprachen verwenden lässt.

Der Zwischenschritt 2 ist — im Vergleich zur direkten Ausgabe — vorteilhaft, da er noch Umformungen des erzeugten Baumes vor der Ausgabe zulässt und insbesondere weil er erlaubt, mit wenig Aufwand die Ausgabe in mehrere Programmiersprachen umzusetzen, indem dafür nur der dritte Schritt angepasst wird. Für diese Studienarbeit beschränkt sich die Ausgabesprache jedoch auf ML, da der Schwerpunkt in der Umwandlung und nicht in der Ausgabe liegt.

Es wurde entschieden, die Anwendung in Form eines Plugins für die Entwicklungsumgebung Eclipse zu realisieren, da dies mehrere Vorteile bietet: Eclipse ist modularisiert aufgebaut, so dass sich neue Funktionen auf einem gut unterstützten Weg integrieren lassen. Für die Entwicklung in der Programmiersprache Java ist im gewöhnlichen Eclipse-Installationsumfang bereits das Plugin „Java development tools“ (JDT) enthalten. Dieses verfügt für die Unterstützung des Benutzers über ein Java-Frontend, so dass lexikalische und syntaktische Analyse, sowie optional auch semantische Analyse in Form von Namens- und Typauflösung, vorhanden sind. Diese Funktionen stehen für die Benutzung in anderen Plugins zur Verfügung und werden daher von Java2Jinja verwendet, wodurch der erste der oben genannten Schritte nicht mehr selbst implementiert werden musste. Desweiteren bietet Eclipse die Möglichkeit, Plugins in Eclipse selbst zu entwickeln und innerhalb der Umgebung eine zweite Eclipse-Instanz für Testen und Fehlerfindung zu starten, in der das

gerade entwickelte Plugin läuft. Daher wurde Eclipse auch als Entwicklungsumgebung für das Plugin benutzt, damit diese Unterstützung des Entwicklungsprozesses nutzbar war.

Die Wahl der Programmiersprache war hiermit auch bereits festgelegt, da Eclipse nur Java für seine Plugins unterstützt. Aufgrund von persönlichen Vorerfahrungen und auch der allgemeinen Bekanntheit im universitären Umfeld, insbesondere in Hinblick auf spätere Weiterentwicklung durch andere Personen, wäre die Wahl aber ohnehin so gefallen.

Das Plugin „CoreC++“, das von Mitarbeitern des selben Lehrstuhls betreut wird, wurde als bereits vorhandenes Beispiel einer ähnlichen Aufgabenstellung betrachtet, aber leider waren die Überschneidungen in den wesentlichen Verarbeitungsteilen so gering, dass die Verarbeitungsschritte 2 und 3 von Grund auf neu implementiert wurden. Lediglich einige Teile der Integration in Eclipse wurden in veränderter Form wiederverwendet.

## 2.2 Systemvoraussetzungen und Installation

Um das Plugin nutzen zu können, muss das System in der Lage sein, Eclipse auszuführen. Die wesentliche Anforderung ist daher, dass für die verwendete Hardware und das Betriebssystem eine Java-Laufzeitumgebung verfügbar sein muss, was die verbreiteten Architekturen x86 und x64 mit den Betriebssystemen Windows, GNU/Linux und Mac OS X einschließt.

Es wird Eclipse in Version 3.5.1 mit dem Plugin „Java development tools“, ebenfalls in Version 3.5.1, zusammen mit Java ab Version 6 benötigt. Wenn Poly/ML ausgeführt werden soll, sollte es Version 5.3 haben, da diese Version erfolgreich getestet wurde. Als Standardbibliothek (siehe 2.4) für die Übersetzung sollte die mitgelieferte angepasste Version von OpenJDK verwendet werden, die auf OpenJDK 6 build b17 basiert.

Das Plugin wird in einem Archiv „Java2Jinja.zip“ geliefert, das zunächst an einen beliebigen Ort extrahiert werden muss. Daraufhin ist die Eclipse-Instanz zu öffnen, in die das Plugin installiert werden soll. Über den Menüpunkt „Help -> Install New Software...“ wird das Fenster „Install“ geöffnet und durch einen Klick auf „Add...“ und dann „Local...“ erscheint ein Ordnerauswahlfenster. In diesem ist der Unterordner „Java2Jinja update site“ des Extraktionsordners zu wählen und zwei mal auf „OK“ zu drücken. Es sollte der Eintrag „Java2Jinja“ in der Liste „Available Software“ erscheinen, der mit einem Haken zu versehen und dann zwei Mal „Next“ zu drücken ist. Schließlich muss noch „I accept the terms of the license agreement“ gewählt und dann mit „Finish“ der Installationsvorgang gestartet werden. Während des Vorgangs sollte eine Warnung mit dem Hinweis, dass unsignierte Software installiert wird, erscheinen, die mit „OK“ bestätigt wird. Die folgende Frage, ob Eclipse neugestartet werden soll, sollte sicherheitshalber mit „Yes“ beantwortet werden.

Wenn ein Java-Projekt nun mit dem Plugin übersetzt werden soll, sind für dieses zunächst über den Menüpunkt „Project -> Properties“ die Projekteigenschaften aufzurufen. Auf der Seite „Java Build Path“ ist dann der Reiter „Libraries“ zu wählen, welcher eine Liste mit

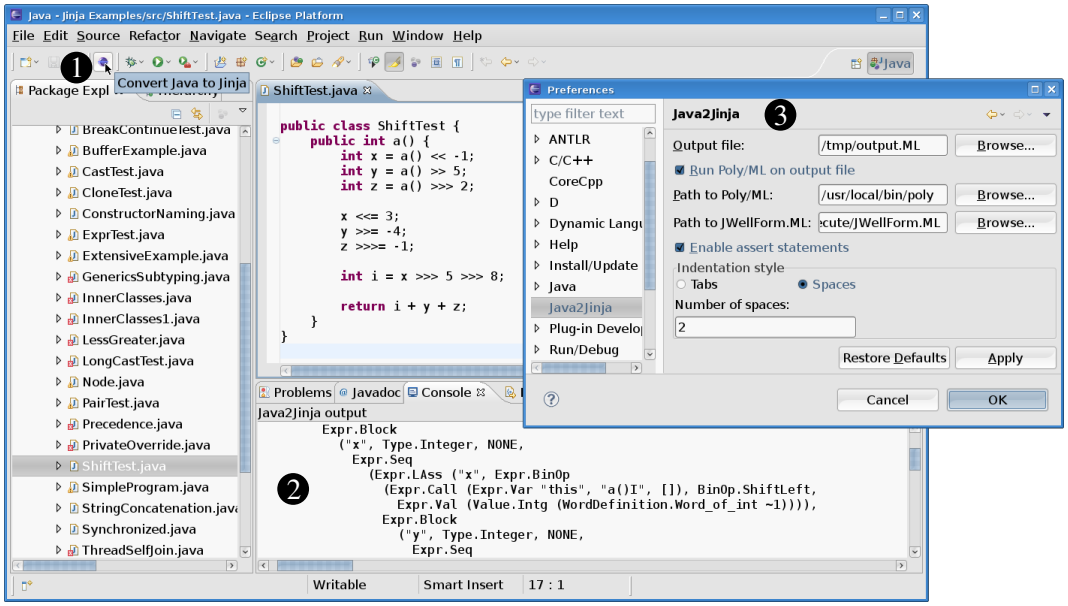



Abbildung 1: Einbettung in die Entwicklungsumgebung

dem Eintrag „JRE System Library“ zeigen sollte. Dieser Eintrag ist auszuwählen und durch einen Klick auf „Remove“ zu entfernen. Nach einem Klick auf „Add External JARs...“ sollte dann die Datei „jdk_classes.jar“ im Unterordner „jdk_classes“ des Extraktionsordners gewählt und durch einen Klick auf „OK“ in den Build Path des Projekts eingefügt werden. Nach einem weiteren Klick auf „OK“ sind die Projekteigenschaften wieder geschlossen und das Plugin kann verwendet werden, indem eine Java-Datei in den Editor geladen und auf den Button „Convert Java to Jinja“ geklickt wird.

## 2.3 Benutzeroberfläche

Java2Jinja bettet sich in die Eclipse-Umgebung an drei Punkten ein (siehe Abbildung 1):

1. Das Kommando „Convert Java to Jinja“ wird sowohl in einer eigenen Gruppe in der Toolbar eingefügt, als auch in das Menü „Run“. Durch Auslösen des Kommandos wird der Übersetzungsvorgang ausgehend von der gerade im Editor geladenen Java-Datei gestartet.

2. In die Konsolenansicht wird eine zusätzliche Konsole eingefügt, um das Übersetzungsergebnis anzuzeigen und optional zusätzlich die Ausgabe der Poly/ML-Sitzung wiederzugeben. Sollte bei der Übersetzung ein Fehler auftreten, wird dieser jedoch nicht über die Konsole ausgegeben, sondern als Eclipse-Fehlermeldung.

3. Die Eclipse-Einstellungen werden um eine zusätzliche Einstellungsseite „Java2Jinja“ erweitert, die es ermöglicht, globale Einstellungen für das Plugin festzulegen:

- Output file: Nach der Übersetzung wird der erzeugte Code in eine Datei ausgegeben und zusätzlich in der Konsole angezeigt. Als Standard für diese Einstellung ist „/tmp/output.ML“ festgelegt, sodass keine bestehenden Dateien überschrieben werden. Jedoch kann der Benutzer den Pfad so anpassen, dass die Ausgabe an eine für ihn günstigere Position geschieht.

- Run Poly/ML on output file: Java2Jinja bietet eine automatische Ausführung von Poly/ML im Anschluss an die Übersetzung an, womit der neu generierte Code gleich auf Wohlgeformtheit getestet werden kann. Die Ausgabe von Poly/ML geschieht dann ebenfalls in die Konsole unterhalb des ausgegebenen Codes. Poly/ML wird so aufgerufen, dass zunächst ausgegeben wird, ob das Programm wohlgeformt ist, wobei im negativen Fall durch lineares Durchlaufen der Klassen alle fehlerhaften Klassen aufgelistet werden und innerhalb dieser Klassen die fehlerhaften Methoden. Diese Option ist standardmäßig aktiviert.

- Path to Poly/ML: Wenn Poly/ML ausgeführt werden soll, muss hier der Pfad zur ausführbaren Datei „poly“ angegeben werden. Sollte dieses Feld leer sein oder auf eine nicht vorhandene Datei verweisen, erhält der Benutzer beim Start der Übersetzung eine Fehlermeldung, wenn die Ausführung von Poly/ML aktiviert ist.

- Path to JWellForm.ML: Die Datei JWellForm.ML enthält den ML-Code, um die Wohlgeformtheit eines Jinja-Programms zu überprüfen. Daher muss Java2Jinja der Pfad zu dieser Datei für die Ausführung von Poly/ML bekannt gemacht werden. Sollte dieses Feld leer sein oder auf eine nicht vorhandene Datei verweisen, erhält der Benutzer beim Start der Übersetzung eine Fehlermeldung, wenn die Ausführung von Poly/ML aktiviert ist.

- Enable assert statements: Diese Option schaltet global die Verarbeitung von `assert`-Statements um. Ist sie aktiviert (Standardeinstellung), wird der Überprüfungscode generiert, sonst wird nur eine Leeroperation eingesetzt. (siehe 4.13)

- Indentation style: Bei der Ausgabe des ML-Codes wird zur besseren Lesbarkeit eine Einrückung durchgeführt, wofür diese Einstellung das Format einer Einrückungsstufe festlegt. Wird „Spaces“ gewählt, gibt „Number of spaces“ die Anzahl der Leerzeichen an, die eine Einrückungsstufe ergeben, also lässt sich mit der Zahl 0 die Einrückung deaktivieren. Wenn „Tabs“ gewählt ist, wird ein Tabulator-Zeichen verwendet. Als Standardeinstellung sind 2 Leerzeichen ausgewählt.

Zu beachten ist, dass für die Entwicklung und den Test des Plugins das Betriebssystem GNU/Linux verwendet wurde, weshalb die voreingestellten Pfade für diese Umgebung vorgesehen sind. Unter anderen Betriebssystemen müssen diese vor Benutzung des Plugins angepasst werden.

## 2.4 Standardbibliothek

Einige Sprachelemente von Java erfordern das Vorhandensein einer Menge von systemrelevanten Klassen, beispielsweise für Strings und Zusicherungen die Klassen java.lang.String und java.lang.AssertionError. Andere Klassen sind für den Aufbau der Klassenhierarchie notwendig und müssen daher sogar in jedem Jinja-Programm vorkommen (siehe 3.4), unter anderem die Klassen java.lang.Object und java.lang.Throwable. Um den Entwicklungsaufwand zu reduzieren und gleichzeitig eine Richtlinie zu haben, welche Sprachelemente von Java2Jinja unterstützt werden müssen, werden diese Klassen nicht von Hand neu implementiert, sondern soweit möglich wird eine bereits existierende, im Quellcode erhältliche Java-Standardbibliothek verwendet: OpenJDK.

Die Bibliothek OpenJDK, erhältlich unter `http://openjdk.java.net/`, enthält freie Implementierungen der meisten Klassen aus der Java Standard Edition API, insbesondere die Pakete java.lang und java.util. Das Paket java.lang enthält unter anderem alle Systemklassen und ist daher zwingend für die Übersetzung erforderlich. Jedoch sind einige Klassen darin nicht direkt mit Java2Jinja verwendbar, da sie `native`-Methoden enthalten oder von solchen abhängen, die vom Jinja-Laufzeitsystem nicht bereitgestellt werden, aber für grundlegende Funktionen benötigt werden. Deshalb wurden die Klassen java.lang.Thread, java.lang.Throwable und java.lang.System teilweise reimplementiert, teilweise im Funktionsumfang reduziert und ersetzen in dieser neuen Fassung die Implementierungen von OpenJDK. Diese an Java2Jinja angepasste Version von OpenJDK wurde in einem JAR-Archiv untergebracht und sollte anstelle der in Eclipse voreingestellten Standardbibliothek in den Classpath des Projekts eingebunden werden, in dem sich die vom Plugin zu verarbeitenden Dateien befinden.

# 3 Jinja

Der Hauptteil dieser Studienarbeit beschäftigt sich mit der Übersetzung von Java-Sprachelementen in den Syntaxbaum einer erweiterten Form der Sprache Jinja, welche Jinja with Threads genannt wird. Die Bestandteile des ursprünglichen Jinja sind im Artikel „A Machine-Checked Model for a Java-Like Language, Virtual Machine and Compiler“ von Klein und Nipkow [KN06] beschrieben, worauf der Artikel „Type Safe Nondeterminism - A Formal Semantics of Java Threads“ von Lochbihler [Loc08] Bezug nimmt und Erweiterungen einführt. Im folgenden werden die Sprachelemente von Jinja with Threads aufgeführt, wobei zwischen denen, die in Jinja bereits vorkommen und denen, die durch Jinja with Threads hinzukommen, unterschieden wird. Um die Lesbarkeit zu verbessern und für die Verwendung in den weiteren Kapiteln, wird für Ausdrücke zunächst nicht die ursprüngliche Jinja-Notation (aus [KN06]) verwendet, sondern die Ausdrucksbezeichner der Ausgabesprache ML von Java2Jinja, welche den Ausdrücken der ursprünglichen Notation, wie in Tabelle 1 gezeigt, zugeordnet werden.

| ML-Bezeichner | Parameter | Jinja-Syntax |
|---|---|---|
| Seq | $e_1, e_2$ | $e_1; e_2$ |
| Cond | $e_1, e_2, e_3$ | `if` $(e_1)$ $e_2$ `else` $e_3$ |
| While | $e_1, e_2$ | `while` $(e_1)$ $e_2$ |
| TryCatch | $e_1, C, V, e_2$ | `try` $e_1$ `catch` $(C\ V)$ $e_2$ |
| Throw | $e$ | `throw` $e$ |
| Block | $V, T, e$ | $\{V : T;\ e\}$ |
| Var | $V$ | `Var` $V$ |
| LAss | $V, e$ | $V := e$ |
| FAcc | $e, F, D$ | $e.F\{D\}$ |
| FAss | $e_1, F, D, e_2$ | $e_1.F\{D\} := e_2$ |
| AAcc* | $e_1, e_2$ | $e_1[e_2]$ |
| AAss* | $e_1, e_2, e_3$ | $e_1[e_2] := e_3$ |
| ALen* | $e$ | $e$`.length` |
| Val | $v$ | `Val` $v$ |
| BinOp | $e_1, bop, e_2$ | $e_1$ *bop* $e_2$ |
| New | $C$ | `new` $C$ |
| NewArray* | $T, e$ | `newA` $T[e]$ |
| Call | $e, M, es$ | $e.M(es)$ |
| Cast | $T, e$ | `Cast` $T$ $e$ |
| InstanceOf* | $e, T$ | $e$ `instanceof` $T$ |
| Synchronized* | $e_1, e_2$ | `sync`$(e_1)$ $e_2$ |

Tabelle 1: Gegenüberstellung von Jinja-Syntax und ML-Bezeichnern für Ausdrücke. Durch Jinja with Threads eingeführte Bezeichner sind mit * gekennzeichnet. Parameterbezeichnungen: $T$ Typ, $v$ Wert, $C$ Klassenname, $M$ Methodenname, $F$ Feldname, $D$ Name der das Feld deklarierenden Klasse, $V$ Variablenname, $e$ Ausdruck, $es$ Ausdrucksliste, *bop* binärer Operator

## 3.1 Typen und Werte

Jinja enthält die primitiven Typen **Boolean**, **Integer** und **Void**, die den Java-Typen `boolean`, `int` und `void` entsprechen. Für Referenztypen ist **Class** und **NT** enthalten, wobei Class der Typ einer Klasseninstanz, NT der Typ des Null-Literals ist und Class den Klassennamen als Parameter hat. Durch Jinja with Threads wird zusätzlich der Typ **Array** eingeführt, der Arrays wie in Java darstellen kann, indem er einen Komponententyp enthält.

Um konstante Werte der primitiven Typen darzustellen, enthält Jinja **Bool** für Boolean-Werte, **Intg** für Integer-Werte und **Unit** für Void-Werte. Unit hat kein Java-Äquivalent, es ist der einheitliche Rückgabewert aller Void-Ausdrücke. Für die Referenztypen hat Jinja die Werte **Addr**, was eine Speicheradresse darstellt, und **Null**, was dem Java-Literal `null` entspricht. Allerdings wird von Java2Jinja nie Addr ausgegeben, da Objekte stets per New erzeugt werden und Java selbst keine Möglichkeit bietet, explizit Speicheradressen anzuge-

ben.

## 3.2 Ausdrücke

In Jinja bestehen Methodenrümpfe nur aus Ausdrücken, es existieren keine Statements wie in Java. Sämtliche Ausdrücke haben einen Rückgabewert, welcher abhängig von den Parametertypen sein kann. Es existiert keine Aneinanderreihung von Anweisungen, sondern diese ergibt sich durch den Ausdruck **Seq**, der einen linken und einen rechten Ausdruck als Parameter hat, von denen zunächst der linke, dann der rechte ausgewertet und der Wert des rechten Ausdrucks zurückgegeben wird. Auch Kontrollflussstrukturen, die in Java durch Statements realisiert werden, sind in Jinja Ausdrücke: **Cond** stellt eine Bedingung dar und ist äquivalent zum funktionalen `if` von Java (Bedingung `?` Then-Ausdruck **:** Else-Ausdruck), daher müssen die Typen von Then- und Else-Ausdruck gleich sein oder in einer Subtyp-Relation stehen. Das Jinja-**While** entspricht dem `while` von Java, wobei das Ergebnis des Rumpf-Ausdrucks nicht verwendet wird und daher beliebigen Typ haben darf, der Rückgabewert ist Unit. Für die Ausnahmebehandlung verfügt Jinja über die Ausdrücke **TryCatch**, welches einem `try` mit genau einer `catch`-Klausel entspricht und **Throw**, welches identisch mit dem `throw` von Java ist. Bei TryCatch müssen die Typen von `try`-Rumpf und `catch`-Rumpf kompatibel sein, da einer von beiden, je nach Ausgang der Auswertung, den Rückgabewert ergibt. Throw gibt Unit zurück, entsprechend seiner Herkunft als Statement.

Lokale Variablendeklarationen werden durch den Ausdruck **Block** ermöglicht, welcher einen Typ für die Variable, einen Namen und optional einen konstanten Wert als Initialisierung hat. Außerdem enthält er einen Ausdruck, der den Gültigkeitsbereich der neu deklarierten Variable darstellt, als nächstes ausgeführt wird und dessen Rückgabewert der Rückgabewert des Block-Ausdrucks ist. Auf lokale Variablen wird mit dem Ausdruck **Var** zugegriffen und mit **LAss** ein Wert zugewiesen, wobei als Parameter der Variablenname gegeben wird. Auf Felder wird entsprechend mit **FAcc** und **FAss** zugegriffen bzw. zugewiesen, allerdings wird nicht nur das Objekt und der Feldname übergeben, sondern auch der Name der Klasse, die das Feld deklariert. Jinja with Threads führt zusätzlich Arrayzugriff und Arrayzuweisung mit den Ausdrücken **AAcc** und **AAss** ein, die als Parameter das Array und den Index des Elements haben. Die Länge eines Arrays wird nicht durch ein Feld im Array-Objekt verfügbar gemacht, sondern Jinja with Threads sieht hierfür **ALen** vor. Zu beachten ist bei den Zugriffsausdrücken, dass sie den Wert der Variablen oder des Feldes zurückgeben, wohingegen die Zuweisungsausdrücke nicht wie in Java den Wert, sondern Unit zurückgeben.

Literale werden nicht direkt durch den Wert realisiert, da dieser kein Ausdruck ist, sondern in einem **Val**-Ausdruck. Im Gegensatz zu Java verfügt Jinja nicht über unäre, sondern nur über binäre arithmetische und logische Operatoren, welche durch **BinOp** mit dem linken Operand, dem Operator und dem rechten Operand als Parameter angegeben werden. Jinja unterstützt ursprünglich nur die Operatoren **Add** (entspricht +) und **Eq** (==),

durch Jinja with Threads kommen zusätzlich die Operatoren **Subtract** (−), **Mult** (⋆), **Div** (/), **Mod** (%), **LessThan** (<), **LessOrEqual** (<=), **GreaterThan** (>), **GreaterOrEqual** (>=), **BinAnd** (&), **BinOr** (|), **BinXor** (^), **ShiftLeft** (<<), **ShiftRightZeros** (>>>) und **ShiftRightSigned** (>>) hinzu. Diese Operatoren können bei Jinja with Threads mit den gleichen Operanden verwendet werden wie ihre Java-Pendants. Zur Klasseninstantiierung verfügt Jinja über den Ausdruck **New**, der sich aber von `new` dadurch unterscheidet, dass er lediglich ein Objekt erzeugt, jedoch nicht dessen Konstruktor aufruft. Daher wird New auch lediglich der Klassenname übergeben. Durch Jinja with Threads kommt **NewArray** für die Instantiierung von Arrays hinzu, wodurch aber lediglich *eine* Array-Dimension erzeugt und keine Initialisierung vorgenommen wird, was beim `new` von Java möglich wäre. Methodenaufrufe werden in Jinja durch **Call** ermöglicht, welches die späte Bindung berücksichtigt und daher im Gegensatz zum Feldzugriff nicht den Namen der deklarierenden Klasse übergeben bekommt. Typumwandlungen sind bei Jinja mit **Cast** nur zwischen Referenztypen möglich, durch Jinja with Threads werden sie für beliebige Typen verfügbar, insofern eine Umwandlung grundsätzlich möglich ist. Dabei werden auch zur Laufzeit entsprechende Exceptions geworfen, wenn ein Cast fehlschlägt. Jinja with Threads ergänzt durch **InstanceOf** zusätzlich ein Äquivalent zum `instanceof` von Java, sowie **Synchronized** als Äquivalent zu `synchronized`, welches als Rückgabewert den des gekapselten Ausdrucks weiterreicht.

## 3.3 Programme

Ein Jinja-Programm wird durch eine Liste von Klassendeklarationen repräsentiert, welche wiederum einen Klassennamen, den Namen der Oberklasse, eine Liste von Felddeklarationen und eine Liste von Methodendeklarationen enthält. Eine Felddeklaration besteht nur aus dem Namen und dem Typ des Feldes, also können Feldinitialisierer nicht direkt aus Java übersetzt werden. Eine Methodendeklaration besteht aus dem Namen der Methode, dem Rückgabetyp und einer Liste von Parametertypen sowie dem Methodenrumpf. Der Methodenrumpf enthält eine Liste von Namen, die den Parametertypen in der gleichen Reihenfolge zugeordnet werden, sowie einen Ausdruck, der beim Aufruf der Methode ausgewertet wird, und dessen Wert dann der Rückgabewert der Methode ist.

## 3.4 Wohlgeformtheitsbedingungen

Damit ein Jinja-Syntaxbaum sinnvoll weiterverarbeitet werden kann, müssen bestimmte Wohlgeformtheitsbedingungen eingehalten werden: Das Jinja-Programm muss die Systemklasse Object und bestimmte System-Exceptions, nämlich NullPointerException, ClassCastException und OutOfMemoryException enthalten. Jinja with Threads fordert zusätzlich die Klassen Throwable, Thread und String und auch die Exceptionklassen ArrayIndexOutOfBoundsException, ArrayStoreException, NegativeArraySizeException, IllegalMonitorStateException, IllegalThreadStateException und CloneNotSupportedException. Sämtliche Klassen in einem Jinja-Programm müssen verschiedene Namen haben und wohlgeformt

sein, was heißt dass ihre Methoden und Felder verschiedene Namen haben und ebenfalls wohlgeformt sein müssen, sowie keine Zyklen im Vererbungsgraph bestehen, Methodenüberschreibung bezüglich der Parametertypen kontravariant, bezüglich der Rückgabetypen kovariant sein und für alle Klassen außer Object muss die Oberklasse existieren. Eine Felddeklaration ist wohlgeformt, wenn ihr Typ existiert, ebenso müssen bei Methodendeklarationen die Parameter- und Rückgabetypen existieren. Weiterhin muss bei Methodendeklarationen die Anzahl von Parametertypen und -namen gleich sein, kein Parametername mehrfach vorkommen, kein Parameter „this“ heißen, alle Variablen vor ihrer Benutzung initialisiert werden und der Ausdruck des Methodenrumpfes wohltypisiert sein. Die beiden letzten Bedingungen führen bei der Übersetzung an verschiedenen Stellen dazu, dass scheinbar unnötige Ausdrücke eingefügt werden müssen, worauf an den entsprechenden Stellen eingegangen wird.

# 4 Unterstützte Sprachkonstrukte

## 4.1 Notation

In diesem Kapitel werden nun die Ansätze vorgestellt, mit denen die eigentliche Übersetzung eines Java-Programms in ein Jinja-Programm umgesetzt wird. Dabei werden drei Arten der Darstellung verwendet, um die Beschreibungen im Text zu ergänzen:

- Abstrakte Syntaxbäume. Dabei entspricht die Reihenfolge der Parameter eines Ausdrucks der Anordnung der Unterknoten im Baum von links nach rechts gelesen. Es werden stets zwei Syntaxbäume gegenübergestellt, von denen der linke die Struktur des Java-Codes darstellt, wobei die Grammatikbezeichnungen aus der Java Language Specification (JLS) [GJSB05] für die Knoten verwendet werden und der rechte Baum die Struktur der Jinja-Übersetzung wiedergibt, wobei die oben angegebenen Bezeichner von ML in den Nichtterminal-Knoten verwendet werden.

- Java-Pseudocode. Da die aufwändigeren Übersetzungen zu recht großen Syntaxbäumen führen, wird dafür eine kompaktere Darstellung in Form von Java-Code verwendet, der jedoch Platzhalter enthält, die im Text erläutert werden. Jedoch sollte diese Darstellung nicht so missverstanden werden, dass Java2Jinja zunächst eine Übersetzung auf Java-Ebene vornähme, tatsächlich findet die Übersetzung stets direkt in Jinja-Ausdrücke statt.

- Jinja-Syntax in der ursprünglichen Schreibweise. Für die Darstellung von Umformungen, die sich aufgrund der von Java abweichenden Sprachstruktur und durch ihre Komplexität nicht durch die ersten beiden Arten sinnvoll präsentieren lassen, wird die ursprüngliche Jinja-Syntax verwendet.

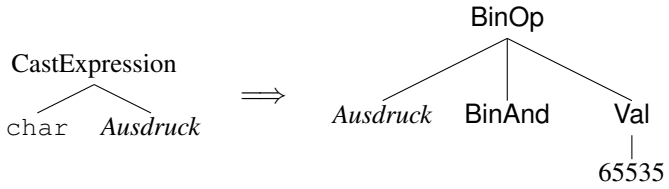
## 4.2 Überblick

In Tabelle 2 sind die durch Java2Jinja unterstützten Java-Sprachelemente aufgelistet. In der Spalte Umsetzung sind bei einigen Elementen direkt ML-Bezeichner angegeben, da die Ähnlichkeit von Java und Jinja für diese eine triviale Umsetzung ermöglicht und es daher wenig Zusatzinformationen liefern würde, diese hier anzugeben. Bei den aufwändigeren Umsetzungen wird auf das entsprechende Kapitel dieses Dokuments verwiesen.

## 4.3 Typen

### 4.3.1 char

Der Typ `char` wird nicht von Jinja unterstützt und daher durch den Jinja-Typ Integer realisiert. Da arithmetische Operationen auf `char`-Werten ohnehin als Ergebnis einen `int`-Wert haben und laut JLS [GJSB05, 4.2.2] vor der Operation nach `int` konvertiert werden, führt dies zu keinen Problemen mit Über- oder Unterläufen. Auch bei der Überladung von Methoden entstehen keine Probleme, da diese von Jinja ohnehin nicht unterstützt wird und daher anders implementiert ist (siehe 4.4.3). Eine implizite Typumwandlung ist in Java von `char` nach `int` möglich [GJSB05, 5.1.2] und wird durch diese Realisierung auch unterstützt. Die umgekehrte Umwandlung ist nicht implizit möglich, für die explizite Umwandlung muss eine Verkürzung der Bitfolge des Werts stattfinden [GJSB05, 5.1.3], was hier durch eine bitweise Und-Operation realisiert wird:




`char`-Literale werden entsprechend als Intg umgesetzt, wobei als Wert der Codepoint des Zeichens verwendet wird.

### 4.3.2 String

Wie in Java werden Strings durch die Klasse `String` dargestellt und daher wird durch die Benutzung eines String-Literals implizit ein String-Objekt erzeugt. Allerdings wird darauf verzichtet, gleiche Stringliterale auf ein gemeinsames String-Objekt abzubilden, da dafür die Methode `String.intern` benutzt werden müsste (siehe JLS [GJSB05, 3.10.5]), welche `native` ist und nicht vom Jinja-Laufzeitsystem bereitgestellt wird. Für die Erzeugung des String-Objekts wird der Konstruktor `String(char[] value)` verwendet, wofür zunächst aus den einzelnen Zeichen des String-Literals ein Array erzeugt wird. (siehe Listing 1)

| **Typ** | **Umsetzung** |
|---|---|
| `void` | Void |
| `boolean` | Boolean |
| `int` | Integer |
| `char` | siehe 4.3.1 |
| Top-Level-Klasse | Class |
| Array | Array |

| **Literal** | **Umsetzung** |
|---|---|
| `boolean`-Literal | Bool |
| `int`-Literal | Intg |
| `char`-Literal | siehe 4.3.1 |
| `String`-Literal | siehe 4.3.2 |
| `null` | Null |
| `this` | siehe 4.5 |

| **Statement** | **Umsetzung** |
|---|---|
| `return` | siehe 4.6 |
| funktionales `if` | Cond |
| `if` | siehe 4.7.1 |
| `switch` | siehe 4.7.2 |
| `while` | While |
| `for` | siehe 4.8.1 |
| `do`/`while` | siehe 4.8.2 |
| `try`/`catch` | siehe 4.12 |
| `finally` | siehe 4.12 |
| `throw` | Throw |
| `assert` | siehe 4.13 |
| `synchronized` | Synchronized |

| **Ausdruck** | **Umsetzung** |
|---|---|
| Variablenzugriff / -zuweisung | Var / LAss |
| Feldzugriff / -zuweisung | FAcc / FAss |
| Arrayzugriff / -zuweisung | AAcc / AAss |
| Array-`length`-Attribut | ALen |
| Methodenaufruf | Call |
| `super`-Methodenaufruf | Call |
| `super`-Konstruktoraufruf | siehe 4.4.3 |
| `this`-Konstruktoraufruf | siehe 4.4.2 |
| `new` (Klasseninstantiierung) | siehe 4.9 |
| `new` (Arrayerzeugung) | siehe 4.10 |
| `cast` | Cast |
| `instanceof` | InstanceOf |

| **Operator** | **Umsetzung** |
|---|---|
| `==,!=,<,<=,>,>=` | BinOp |
| `+,-,*,/,%` | BinOp |
| `&,|,^,<<,>>,>>>` | BinOp |
| Unär: +, −, ∼, !, ++, −− | siehe 4.11.1 |
| Zuweisungsoperatoren: +=, etc. | siehe 4.11.2 |
| `&&, ||` | siehe 4.11.3 |
| Stringkonkatenation | siehe 4.11.4 |

| **Modifikator** | **Umsetzung** |
|---|---|
| `synchronized` | siehe 4.4.3 |
| `abstract` | siehe 4.4.3 |
| `private` | siehe 4.4.3 |
| `native` | siehe 4.4.3 |
| `static` | siehe 4.4.4 |

Tabelle 2: Überblick über die unterstützten Sprachelemente von Java

```
String s = "abcde";
//wird übersetzt als:
char[] temp0 = new char[]{'a', 'b', 'c', 'd', 'e'};
String s = new String(temp0);
```

Listing 1: Umsetzung eines String-Literals

## 4.4 Deklarationen

### 4.4.1 Klassendeklarationen

Java-Klassen lassen sich weitgehend direkt auf Jinja-Klassen abbilden, indem die `extends`-Klausel für die Bestimmung der Oberklasse verwendet wird, falls sie vorhanden ist. Sonst wird als Oberklasse java.lang.Object angenommen, wie in Java üblich. Nichtstatische Felder und Methoden werden direkt zu den entsprechenden Mengen hinzugefügt, für statische Member siehe 4.4.4. Klassen, die keinen explizit angegebenen Konstruktor enthalten, werden um einen Standardkonstruktor ergänzt, wie in der JLS [GJSB05, 8.8.9] angegeben.

Klassennamen können nicht ohne weiteres einfach übernommen werden, da in Jinja alle Klassen verschieden heißen müssen, jedoch in Java Klassen in verschiedenen Paketen den gleichen Namen haben dürfen. Daher werden vollqualifizierte Klassennamen für die Jinja-Klassen verwendet, jedoch wird dafür nicht die übliche Schreibweise mit '.', sondern die Namenskonvention aus der Java Virtual Machine Specification (JVMS) [LY99, 4.2] verwendet (Trennung durch '/'), um eine Konsistenz mit der Benennung von Klassentypen in Methodennamen zu erreichen (siehe 4.4.3).

### 4.4.2 Felddeklarationen

Nichtstatische Felder werden direkt in Jinja-Felder in den Klassen, wo sie auch in Java deklariert werden, übersetzt, indem Namen und Typen festgelegt werden. Bei Feldern können die Namen direkt übernommen werden, da es auch in Java Felder des gleichen Namens nur einmal geben darf.

Feldinitialisierer lassen sich in Jinja hingegen nicht direkt darstellen, deshalb müssen sie in die entsprechenden Methodenrümpfe als Zuweisungen eingefügt werden, so dass sie bei der Erzeugung einer Klasseninstanz ausgeführt werden. Nach der JLS [GJSB05, 8.3.2.2] können von Feldinitialisierern auch die Schlüsselworte `this` und `super` benutzt werden, also können Feldinitialisierer nicht dadurch realisiert werden, dass sie zwischen New und dem Aufruf des Konstruktors eingefügt werden, da dies außerhalb des Gültigkeitsbereichs dieser Schlüsselworte zum neu erzeugten Objekt ist. Daher werden diese Zuweisungen in die Konstruktoren einer Klasse eingefügt, wobei sicherzustellen ist, dass kein Feldinitialisierer mehrfach ausgeführt wird, da diese auch Seiteneffekte haben können. Daher werden bei Konstruktoren, die einen anderen Konstruktor per `this`-Konstruktoraufruf verwenden, keine Feldinitialisierungszuweisungen eingefügt, da ja der andere Konstruktor dann schon die Initialisierung enthält. Die Reihenfolge der Zuweisungen entspricht der Reihenfolge der Felddeklarationen, da Feldinitialisierer auf vorher deklarierte Felder verweisen dürfen [GJSB05, 8.3.2.3] und diese daher bereits vorher initialisiert worden sein müssen. Die Initialisierer werden nach dem expliziten oder impliziten Aufruf des Oberklassenkonstruktors eingefügt, damit die Felder der Oberklasse ebenfalls bereits initialisiert sind und von den

Unterklassenfeldinitialisierern benutzt werden können. (siehe Listing 2)

```
class C extends Oberklasse {
    Typ f = Initialwert;
    Typ2 f2 = Initialwert2;

    C(int x) {
        super(x);
        Konstruktorrumpf;
    }
    C() {
        this(0);
    }
}
//wird übersetzt als:
class C extends Oberklasse {
    Typ f;
    Typ2 f2;

    C(int x) {
        super(x);
        f = Initialwert;
        f2 = Initialwert2;
        Konstruktorrumpf;
    }
    C() {
        this(0);
        //keine Feldinitialisierung
    }
}
```

Listing 2: Übersetzung von Feldinitialisierern

### 4.4.3 Methodendeklarationen

Im Gegensatz zu Feldern muss bei Methoden eine Dekoration des Namens stattfinden, um die Überladung von Methoden zu unterstützen. Da in Jinja Methoden nur anhand des Namens und nicht der vollständigen Signatur unterschieden werden, wird von Java2Jinja der Name um Parameterinformationen ergänzt, indem das selbe Schema angewendet wird, das auch intern in der Java Virtual Machine benutzt wird (siehe [LY99, 4.3]): Primitive Typen werden durch einzelne Buchstaben ausgedrückt ('I' für `int`, 'C' für `char`, 'Z' für `boolean` und 'V' für `void`, Klassen werden durch den Klassennamen in der Form die in 4.4.1 beschrieben ist, wobei ein 'L' vorangestellt und ein ';' nachgestellt wird, ausgedrückt und für Arrays wird pro Dimension ein '[' vorangestellt. Diese Typen werden direkt hintereinander innerhalb von runden Klammern an den Methodennamen angehängt und hinter den runden Klammern wird nach selbem Schema der Rückgabetyp angegeben. Zum Beispiel wird aus `void methode(int a, char[][] b, String[] c)` dann `methode(I[[C[Ljava/lang/String;)V`.

Diese Veränderung genügt aber noch nicht, um überschriebene Methoden aufrufen zu können, wie es beispielsweise durch den `super`-Methodenaufruf möglich ist. Dafür muss sichergestellt sein, dass in einer Vererbungshierarchie alle Methoden über verschiedene Namen erreichbar sind und somit muss der Name der deklarierenden Klasse in den dekorierten Methodennamen einfließen. Dies wurde realisiert, indem der Klassenname in der Schreibweise aus 4.4.1 vorangestellt und mithilfe des Zeichens '∼' separiert wird. Würde die oben genannte Beispielmethode in der Klasse java.lang.String stehen, ergäbe sich hiermit die Zeichenfolge `java/lang/String∼methode(I[[C[Ljava/lang/String;)V`. Eine solche Darstellung kann allerdings nicht allein für die Repräsentation einer Methode verwendet werden, da dann das Überschreiben unmöglich würde, da Jinja nur eine Überschreibungsrelation zwischen Methoden mit gleichem Namen herstellt. Um sowohl Aufruf überschriebener Methoden als auch Überschreiben zu ermöglichen, wird für jede normale Methode eine virtuelle Jinja-Methode geführt, deren Name nach dem ersten Schema ohne den Namen der deklarierenden Klasse entsteht und weiterhin eine nicht-virtuelle, also statisch gebundene Methode, die nach dem zweiten Schema entsteht. Die virtuelle Methode ist jedoch keine Duplikation der nicht-virtuellen Methode, sondern führt lediglich einen Aufruf der anderen Methode aus und reicht den Rückgabewert weiter. Sämtliche gewöhnlichen Methodenaufrufe verwenden die virtuelle Methode, lediglich der `super`-Methodenaufruf und unten genannte Sonderfälle verweisen direkt auf die statisch gebundene Methode.

Konstruktoren werden als Methoden realisiert, da Jinja hierfür kein besonderes Mittel vorsieht und konsistent zur JVMS [LY99, 3.9] mit dem Namen „<init>“ bezeichnet, wobei als Rückgabetyp `void` verwendet wird und die Dekoration wie gewöhnlich stattfindet. Jedoch wird nur eine statisch gebundene Methode generiert, da bei Konstruktoren ohnehin kein Überschreiben möglich ist und die aufgerufene Methode bereits zur Übersetzungszeit feststeht. Bei Konstruktoren wird außerdem überprüft, ob der Rumpf mit einem expliziten Konstruktoraufruf beginnt, damit bei Nichtvorhandensein automatisch ein Aufruf des Oberklassenkonstruktors ohne Argumente vorangestellt werden kann, entsprechend der JVMS [LY99, 2.12].

Der Modifikator `synchronized` wird als `synchronized`-Statement realisiert, indem eine so deklarierte Methode wie eine Methode mit `synchronized`-Block um den Rumpf behandelt wird, in Übereinstimmung mit der JLS [GJSB05, 8.4.3.6]:

```
synchronized void methode() {
    Rumpf;
}
//wird übersetzt als:
void methode() {
    synchronized(this) {
        Rumpf;
    }
}
```

Methoden, für die kein Rumpf definiert ist, weil sie als `abstract` deklariert sind, können eigentlich nie ausgeführt werden, da sie stets von einer nicht-abstrakten Methode überschrieben sein müssen. Sie können aber nicht einfach bei der Übersetzung übergangen wer-

den, da die Typprüfung die Existenz einer entsprechenden Methode in der abstrakten Klasse erfordert, jedoch bietet Jinja keine Möglichkeit eine Methode als abstrakt zu kennzeichnen. Für solche Methoden wird daher eine gewöhnliche Methode erzeugt, allerdings wird im generierten Rumpf unmittelbar eine UnsupportedOperationException geworfen, um in dem Fall, der eigentlich nie eintreffen sollte, auf den Fehler hinzuweisen. Da Jinja in jeder Nicht-`void`-Methode einen Rückgabewert erwartet, auch wenn die Methode nie auf normalem Wege verlassen werden kann, wird noch ein redundanter Rückgabewert generiert:

```
abstract Typ methode();
//wird übersetzt als:
Typ methode() {
    throw new UnsupportedOperationException();
    return Dummy-Wert entsprechend Typ;
}
```

Zugriffsmodifikatoren haben in Jinja grundsätzlich keine Entsprechung und werden von Java2Jinja ignoriert, mit Ausnahme von `private`. Dies ist notwendig, da private Methoden es möglich machen, Methoden mit gleicher Signatur in einer Vererbungshierarchie zu verwenden, ohne dass ein Überschreiben stattfindet (siehe JLS [GJSB05, 8.4.8.3]). Somit findet bei privaten Methoden auch keine späte Bindung statt, weshalb von Java2Jinja nur eine statisch gebundene, aber keine virtuelle Methode erzeugt wird. Aufrufe von privaten Methoden verweisen direkt auf die statisch gebundene Methode.

Der Modifikator `native` wird derart unterstützt, dass für die so deklarierte Methode keine Jinja-Methode generiert wird. Dies hat den Zweck, Methoden die von der Jinja-Laufzeitumgebung bereitgestellt werden und daher nicht in der Ausgabe vorkommen dürfen, zumindest für die Abhängigkeitsauflösung der Methodenaufrufe verfügbar zu haben.

### 4.4.4 Statische Felder und Methoden

Statische Felder und Methoden werden direkt in Form der `main`-Methode, aber auch innerhalb von Methoden der Standardbibliothek, die von außen betrachtet nicht statisch sind, häufig verwendet und sind deshalb für die Benutzbarkeit der unterstützten Untermenge von Java von großer Bedeutung. Daher wurde entschieden, trotz fehlender Unterstützung durch Jinja diese Sprachelemente verfügbar zu machen. Das Grundproblem besteht darin, dass es in Jinja keinerlei globale Objekte gibt, die eine Speicherung von statischen Informationen erlauben, sondern nur die Objekte, die vom Programm erzeugt und innerhalb von Feldern und lokalen Variablen gespeichert, also eigentlich für jede Klasseninstanz individuell sind. Die Idee liegt darin, ein Objekt global einmalig zu erzeugen und dann überall verfügbar zu machen, wo auf statische Felder und Methoden zugegriffen werden kann. Dazu muss eine zusätzliche Klasse für die globalen Daten erzeugt und sämtliche Klassen des Programms erweitert werden. Um die Invasivität dieser Veränderung gering zu halten, wird das Objekt nicht in lokalen Variablen gespeichert, denn dann müsste jede Methode einen zusätzlichen Parameter erhalten, was dann auch weitere Eingriffe bei der Wohlgeformtheitsprüfung bedeuten würde. Stattdessen wird jede Klasse um ein Feld erweitert, das die Referenz auf

diese einmalige Klasseninstanz speichert, welche bei jeder Erzeugung eines neuen Objekts in dessen entsprechendes Feld übertragen wird.

Die globale Klasse wird von Java2Jinja mit dem Namen „∼GlobalStatics“ im Default-Package erzeugt, wobei das Zeichen ’∼’ bewirkt, dass keine Kollision mit einer benutzerdefinierten Klasse geschehen kann. Sie ist direkt von Object abgeleitet und enthält sowohl für sämtliche statischen Felder als auch statischen Methoden des Programms eine nichtstatische Entsprechung: Eine statische Methode wird, genau wie die statisch gebundenen Methoden gewöhnlicher Klassen, durch das zweite Dekorationsschema aus 4.4.3, welches auch den qualifizierten Namen der deklarierenden Klasse enthält, benannt, wobei mit deklarierender Klasse hier die Klasse gemeint ist, die die statische Methode ursprünglich enthielt und nicht der Name der globalen Klasse. Statische Felder werden in Anlehnung an die Methodendekoration ebenfalls durch Voranstellen des qualifizierten Klassennamens und Abtrennung durch ’∼’ dargestellt, so dass sichergestellt ist, dass keine Kollisionen zwischen Feldern verschiedener Klassen entstehen.

Jede Klasse des Programms, die direkt von Object abgeleitet ist, erhält das zusätzliche Feld „∼statics“ vom Typ „∼GlobalStatics“, wodurch auch hier eine Kollision mit anderen Feldnamen vermieden wird. Die Klasse java.lang.Object selbst darf keine Felder enthalten und erhält deshalb dieses Feld nicht, ebenso wie Klassen die nicht direkt von Object ableiten, da für diese bereits in einer Oberklasse das Feld deklariert ist. Auch die Klasse „∼GlobalStatics“ selbst erhält das Feld, obwohl dies letztlich dort nur mit dem Wert von `this` identisch ist. Jedoch wird auf diese Weise eine Vereinheitlichung des Zugriffs von Methodenrümpfen aus erreicht, sodass bei der Übersetzung nicht unterschieden werden muss, ob der Zugriff innerhalb einer statischen oder einer nicht-statischen Methode steht. (siehe Listing 3)

Zugriffe und Zuweisungen auf statische Felder geschehen, indem das Feld „∼statics“ in der umgebenden Klasse als Objekt des Zugriffs verwendet wird und der Name wie oben beschrieben angepasst wird. Ebenso werden Methodenaufrufe mit angepasstem Methodennamen und Bezugsobjekt umgesetzt. Für den Zugriff auf das Feld „∼statics“ ist noch zu beachten, dass die deklarierende Klasse eventuell nicht die Klasse selbst, sondern eine Oberklasse sein kann, weshalb diese durch Aufsteigen in der Klassenhierarchie ermittelt wird.

Initialisierer für statische Felder werden in den Konstruktor der globalen Klasse eingefügt, welcher wie ein gewöhnlicher Konstruktor im Zuge der Erzeugung der globalen Klasseninstanz aufgerufen wird. Diese Instanz wird in den main-Methoden eines Programms erzeugt, noch bevor der Rumpf der main-Methode ausgeführt wird. Da ein Programm nur mit einer seiner main-Methoden gestartet wird, bedeutet das keine Mehrfacherzeugung der Instanz. Dies ist eine Abweichung von den Vorgaben der JLS [GJSB05, 12.4.1], welche fordert, dass Klassen erst initialisiert werden, wenn bestimmte Ereignisse, wie z.B. die erste Instantiierung einer Klasse, eintreffen. Von Java2Jinja wird keine Reihenfolge für die Klasseninitialisierung garantiert, da diese in der aktuellen Implementierung von der Abhängigkeitsauflösung abhängt.

```
class C {
    static int methode(Parameter) {return feld;}
    static int feld = 0;
    void methode2() {methode(Argumente);}
}
//wird übersetzt als:
class C {
    void methode2() {~statics.C~methode(Argumente);}
    ~GlobalStatics ~statics;
}
class ~GlobalStatics {
    ~GlobalStatics() {~statics.C~feld = 0;}
    int C~methode(Parameter) {return ~statics.C~feld;}
    int C~feld;
    ~GlobalStatics ~statics;
}
```

Listing 3: Beispiel für die Übersetzung von statischen Feldern und Methoden

```
if ((Var this).~statics{~GlobalStatics} = null)
   (Var this).~statics{~GlobalStatics} := new ~GlobalStatics;
   ((Var this).~statics{~GlobalStatics}).~statics{~GlobalStatics} :=
      (Var this).~statics{~GlobalStatics};
   ((Var this).~statics{~GlobalStatics}).~GlobalStatics~<init>()V()
else
   unit;
Rumpf von main
```

Abbildung 2: Initialisierung der globalen Klasse am Beginn einer main-Methode

Zu Beginn einer main-Methode wird geprüft, ob das Feld für die globale Klasseninstanz bereits gesetzt ist, was nur beim ersten Aufruf nicht der Fall ist. Sollte es sich um den ersten Aufruf handeln, wird die globale Klasse instantiiert und dem „∼statics“-Feld des Objekts der main-Methode, welches nicht das eigentliche globale Objekt ist, zugewiesen, sowie dem „∼statics“-Feld der Instanz selbst, damit im anschließend aufgerufenen Konstruktor der Zugriff auf die statischen Member bereits möglich ist. (siehe Abbildung 2)

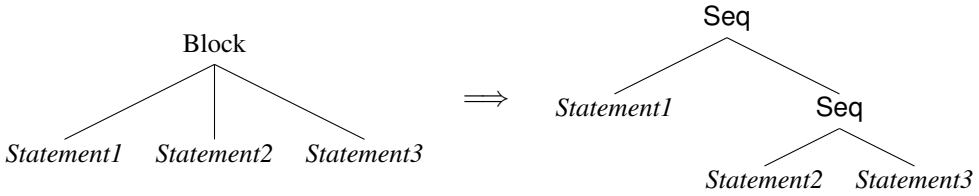
Von der main-Methode ausgehend wird die globale Klasseninstanz übertragen, indem bei jeder Klasseninstantiierung (siehe 4.9) zwischen Erzeugung des Objekts und Aufruf des Konstruktors noch die Zuweisung an das „∼statics“-Feld eingefügt wird, damit innerhalb des Konstruktors bereits statische Member verwendet werden können. Da beim Feldzugriff/-zuweisung die deklarierende Klasse des Feldes angegeben werden muss, ist bei dieser Zuweisung für das neu erzeugte Objekt und für das aktuelle `this`-Objekt die entsprechende Klasse zu bestimmen, hier als „Klasse S“ und „Klasse T“ bezeichnet (siehe Abbildung 3).

{temp0 : *Klassenname*;
  temp0:=`new` *Klassenname*;
  (`Var` temp0).∼statics{*Klasse S*}:=(`Var` this).∼statics{*Klasse T*};
  (`Var` temp0).*Konstruktorname*(*Argumente*);
  `Var` temp0
}

Abbildung 3: Erweiterung der Klasseninstantiierung um die Zuweisung von ∼statics

## 4.5 Blöcke und lokale Variablen

In Java bestehen Blöcke aus einer Sequenz einer beliebigen Anzahl von Statements, die alle auf der gleichen Ebene stehen und einen gemeinsamen Gültigkeitsbereich haben. Jinja hingegen bietet für eine sequentielle Ausführung nur den Ausdruck Seq, welcher genau zwei Unterausdrücke enthält. Daher werden Blöcke mit mehr als zwei Statements realisiert, indem für alle Statements außer das letzte ein Seq-Ausdruck erzeugt wird, welcher als linken Unterausdruck die Repräsentation dieses Statements erhält und als rechten Unterausdruck den Seq-Ausdruck des Nachfolgers. Das letzte Statement wird direkt als rechter Unterausdruck des innersten Seq-Ausdrucks eingefügt. Wenn ein Block nur aus einem Statement besteht, ist der entsprechende Ausdruck direkt die Realisierung des Blocks, wohingegen ein leerer Block durch das Literal Unit ausgedrückt wird.




Lokale Variablen sind in Jinja nicht Statements, die an einer beliebigen Stelle in einem Block vorkommen können, sondern sie werden durch den Ausdruck Block definiert und erzeugen direkt einen neuen Gültigkeitsbereich. Alle Statements, die in einem Block auf eine Variablendeklaration folgen, werden als Unterausdruck in den Block-Ausdruck eingefügt, da sie alle im Gültigkeitsbereich der Variablen liegen. Initialisierer können direkt in einem Block-Ausdruck eingetragen werden, wenn es sich um Literal-Werte handelt, sonst muss eine zusätzliche Zuweisung LAss verwendet werden. Diese beiden Fälle sind in Abbildung 4 beispielhaft enthalten.

Der Ausdruck `this`, dem in Java ein spezielles Schlüsselwort zugeordnet ist, wird in Jinja als lokale Variable mit dem Namen „this“ behandelt. Diese wird nie explizit deklariert, sondern stets implizit in die Umgebung eines Methodenrumpfes eingefügt.

*Statements*;
{x : `Integer` := 0;
  {y : `Boolean`;
    y := `Var` x > 1;
    *Statements*
}}

Abbildung 4: Übersetzung des Blocks {*Statements*; int x = 0; boolean y = x > 1; *Statements*;}

## 4.6 Rücksprung

Die `return`-Anweisung nimmt in Java und ähnlichen Programmiersprachen eine Doppelrolle ein, indem sie einerseits eine Kontrollflussverzweigung bewirkt und andererseits bei Funktionen mit Rückgabetyp den Rückgabewert entgegennimmt. Die zweite Aufgabe lässt sich in Jinja dadurch realisieren, dass das letzte Statement eines Blocks, also in der Übersetzung der rechte Ausdruck des innersten Seq-Knotens, den Rückgabewert des Blocks ergibt. So können `return`-Statements, die direkt am Ende eines Methodenrumpfs stehen, umgesetzt werden, indem die `return`-Anweisung direkt durch den enthaltenen Ausdruck ersetzt wird.

Problematischer ist die Umsetzung des anderen Kontrollflussaspekts der `return`-Anweisung, da Jinja keine direkte Unterstützung dafür bietet. Theoretisch lässt sich jedes Programm mit Sprunganweisungen, wie von Böhm und Jacopini festgestellt [BJ66, 2.], in ein äquivalentes Programm übersetzen, welches nur strukturierte Schleifen und Bedingungen enthält, die sich mit Jinja wieder übersetzen ließen. Jedoch führt diese sogenannte „GOTO-Elimination“ unter Umständen zu einer kaum noch verständlichen Codestruktur und ist aufwändig, weshalb dies den Rahmen dieser Studienarbeit gesprengt hätte.

Stattdessen bleibt als einzige Möglichkeit, das Verlassen eines Blocks zu erwirken, eine Zweckentfremdung der Ausnahmebehandlung: Es wird eine Klasse „∼ReturnException“ generiert, die von java.lang.Throwable abgeleitet ist und keine Member enthält. Für jede `return`-Anweisung, die sich nicht direkt am Ende eines Methodenrumpfes befindet, wird eine Instanz dieser Klasse erzeugt und als Ausnahme geworfen. Zudem wird der Methodenrumpf mit einem TryCatch-Ausdruck umgeben, der diese Exception auffängt, sodass die Methode normal verlassen werden kann.

Um einen Rückgabewert übermitteln zu können, wird bei Nicht-`void`-Methoden zusätzlich am Beginn der Methode eine lokale Variable „∼retval“ deklariert, die als Typ den Rückgabetyp der Methode hat. Diese wird zudem redundanterweise mit einem beliebigen Wert initialisiert, da für die Wohlgeformtheit jede Variable vor ihrer Verwendung initialisiert worden sein muss (siehe 3.4) und die Überprüfung dieser Bedingung nicht so präzise ist, dass erkannt wird, dass dies auch ohne der Fall wäre. Denn bei jedem `return`-Statement wird nun vor dem Werfen der Exception noch diese Variable auf den Rückgabewert gesetzt und

```
int methode(boolean b) {
    if(b) return 1;

    Anweisungen;
    return 2;
}
```

**Umsetzung des Methodenrumpfs in Jinja-Syntax:**

{∼retval : `Integer` := 0;
  `try`
    `if`(`Var` b) (∼retval := 1; `throw new` ∼ReturnException) `else` *unit*;
    *Anweisungen*;
    `Val` 2
  `catch`(∼ReturnException temp0)
    `Var` ∼retval
}

Abbildung 5: `return`-Statements werden abhängig von ihrer Position unterschiedlich übersetzt

im Catch-Ausdruck des umgebenden TryCatch wird der Wert der Variablen zurückgegeben, so dass er den Rückgabewert der Methode ergibt. Wenn die Methode kein zusätzliches `return` am Ende ihres Rumpfes enthält, dass auf dem normalen Weg einen Rückgabewert liefert, wird auch an das Ende des Try-Ausdrucks noch ein Zugriff auf die Variable „∼retval“ angehängt, obwohl dieser nie erreicht würde, damit der Typ des Try-Ausdrucks und des Catch-Ausdrucks gleich ist.

Zu beachten ist, dass `return`-Anweisungen, die unmittelbar am Ende eines Methodenrumpfes stehen, weiterhin auf die einfache Art ohne Exception und Zuweisung realisiert werden und dass nur dann die Variable „∼retval“ und der `TryCatch`-Ausdruck erzeugt werden, wenn sie tatsächlich in Verwendung sind. So wird vermieden, dass Methoden, für die keine solche Behandlung notwendig wäre, unnötig vergrößert werden. In Abbildung 5 ist ein Beispiel angegeben, welches beide Umsetzungsarten für das `return`-Statement zeigt.

## 4.7 Bedingungen

### 4.7.1 if

Das `if`-Statement lässt sich fast direkt mit dem Cond-Ausdruck umsetzen. Bedingung und Then-Statement ergeben stets die entsprechenden Unterausdrücke von Cond. Allerdings muss bei nicht vorhandenem `else`-Statement der entsprechende Unterausdruck von Cond auf ein Unit-Literal gesetzt werden (siehe Abbildung 6), sonst auf den Ausdruck, der sich aus dem `else`-Statement ergibt.

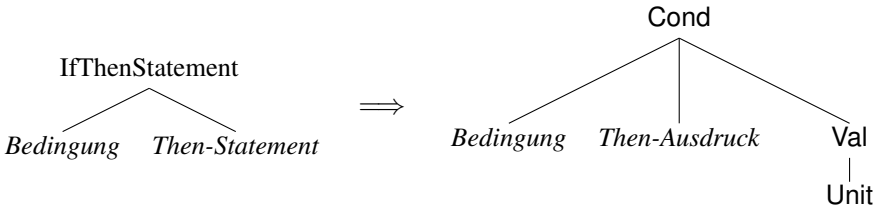



Abbildung 6: Umsetzung eines `if`-Statements ohne `else`

Für die Verarbeitung von Then- und Else-Statement muss beachtet werden, dass die sich ergebenden Ausdrücke beide vom Typ Void sein müssen, da der Cond-Ausdruck die Gleichheit erfordert. Wie dies sichergestellt wird, ist bei den entsprechenden Sprachelementen, wo verschiedene Typen möglich sind (siehe beispielsweise 4.9), angegeben.

### 4.7.2 switch

Wenn ein `switch`-Statement nur aus Fällen besteht, die stets durch ein `break` abgeschlossen werden, mit Ausnahme des letzten Falls, entspricht das `switch`-Statement einer Kaskade von `if`-Bedingungen. Jedoch wird durch Weglassen des `break`-Statements der Kontrollfluss komplexer, so dass die Realisierung anders geschehen muss: Da Jinja keine expliziten Sprünge unterstützt, werden zusätzliche boolesche Variablen benötigt, die wiedergeben, ob ein Durchfallen stattfindet (also kein `break`-Statement einen Fall abschließt) und ob bereits ein Fall ausgeführt wurde, so dass der `default`-Fall nicht mehr ausgeführt wird. So kann dann mithilfe dieser Informationen der Kontrollfluss durch `if`-Statements modelliert werden, indem bei gewöhnlichen Fällen vor der eigentlichen Bedingung die fallthrough-Variable abgefragt wird, beim `default`-Fall zusätzlich die `default`-Fall-Variable.

Innerhalb eines Falles wird zunächst die `default`-Fall-Variable auf `false` gesetzt, da dieser nicht mehr ausgeführt werden soll, wenn ein anderer Fall bereits ausgeführt wurde. Die fallthrough-Variable wird auf `true` gesetzt, damit im Falle eines nicht vorhandenen `break`-Statements ein Durchfallen stattfindet. Entsprechend führt ein `break`-Statement dazu, dass die Variable auf `false` zurückgesetzt wird, wodurch die vorherige Zuweisung von `true` eigentlich redundant wird. Allerdings wird durch diese Redundanz die Verarbeitung vereinfacht und bleibt aus Zeitgründen in dieser Studienarbeit bestehen, sodass dies eine Möglichkeit für zukünftige Optimierungen der Übersetzung ist. Der Switch-Ausdruck muss in eine temporäre Variable zwischengespeichert werden, damit er nicht mehrfach ausgewertet wird. Diese Variable hat stets den Typ `int`, da Java2Jinja kein Unboxing unterstützt und deshalb für den Ausdruck nur die Typen `char` und `int` möglich sind. Aus dem gleichen Grund ist auch keine Abfrage des Ausdrucks auf `null` notwendig (vgl. JLS [GJSB05, 14.11]), da `null` nur durch die Verwendung der Box-Klassen als Ergebnis des Ausdrucks vorkommen könnte. (siehe Listing 4)

```
switch(Ausdruck) {
    case Fall1:
        Statements1;
    case Fall2:
        Statements2;
        break;
    default:
        Default-Statements;
}
//wird übersetzt als:
boolean defaultCase = true;
boolean fallthrough = false;
int temp0 = Ausdruck;
if(fallthrough || temp0 == Fall1) {
    defaultCase = false;
    fallthrough = true;
    Statements1;
}
if(fallthrough || temp0 == Fall2) {
    defaultCase = false;
    fallthrough = true;
    Statements2;
    fallthrough = false;
}
if(fallthrough || defaultCase) {
    Default-Statements;
}
```

Listing 4: Übersetzung des `switch`-Statements

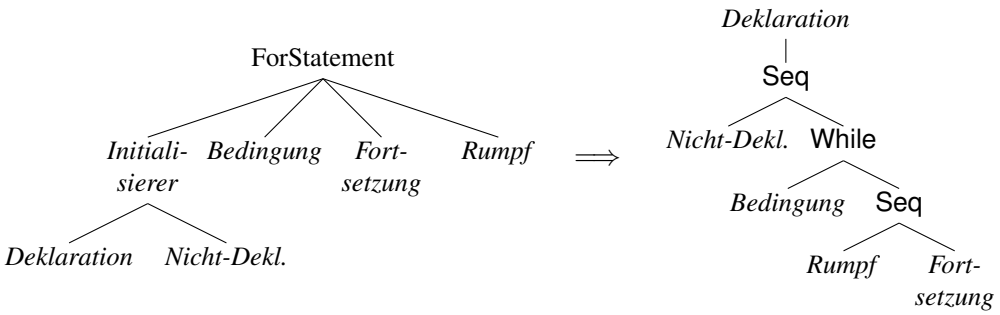
## 4.8 Schleifen

### 4.8.1 for

Die `for`-Schleife wird auf eine `while`-Schleife zurückgeführt. Die Bedingung wird beibehalten, wenn sie existiert. Ist in der `for`-Anweisung an dieser Stelle eine Lücke, wird konstant der Wert `true` als Bedingung eingesetzt. Die Fortsetzungsanweisungen werden per Seq im Anschluss an den Rumpf ausgeführt. Da sie keine lokalen Variablen des Rumpfes benutzen können, müssen sie nicht innerhalb von diesem eingefügt werden. Die Schleifeninitialisierer jedoch sind Anweisungen, die lokale Variablen deklarieren können, die innerhalb der Schleife verwendet werden. Daher können sie nicht einfach per Seq vor den While-Ausdruck gesetzt werden, sondern müssen diesen umschließen. Solche Deklarationen werden in Block-Ausdrücke umgesetzt, die den nächsten Initialisierer bzw. den While-Ausdruck enthalten. Die Nicht-Deklarationen unter den Initialisierern werden per Seq mit dem nächsten Initialisierer bzw. dem While-Ausdruck verkettet:

```
do {
    Rumpf;
}
while (Bedingung);
//wird übersetzt als:
boolean temp0 = true;
while(temp0) {
    Rumpf;
    temp0 = Bedingung;
}
```

Listing 5: Umsetzung des do-Statements




### 4.8.2 do

Die `do`-Schleife führt eine Auswertung der Bedingung erst nach dem Schleifenrumpf aus. Daher wird in der Umsetzung eine temporäre Variable verwendet, die mit `true` initialisiert ist und die am Ende des Schleifenrumpfes auf den neuen Wert der Bedingung gesetzt wird. Als Bedingung des While-Ausdrucks wird dann ein Zugriff auf diese Variable verwendet (siehe Listing 5).

### 4.8.3 break und continue

Um `break` und `continue` unterstützen zu können, wird das gleiche Prinzip wie bei `return` angewendet. Da Jinja keine Kontrollstrukturen außer Cond, While und Seq anbietet, wird die Ausnahmebehandlung zweckentfremdet: Es werden Exceptionklassen je für `break` („∼BreakException“) und für `continue` („∼ContinueException“) generiert und zusammen mit den benutzerdefinierten Klassen ausgegeben. Diese sind direkt von Throwable abgeleitet, da Exception-Klassen laut JLS [GJSB05, 11.1] stets eine Unterklasse von Throwable sein müssen und enthalten beide ein Feld depth, um die Anzahl der zu verlassenden Schleifen zu speichern, damit auch `break` und `continue` mit Labels unterstützt werden können. Wird innerhalb einer Schleife eine `break`-Anweisung benutzt, so wird dies durch das Werfen einer ∼BreakException realisiert, bei der das depth-Attribut entspre-

```
throw {temp0 : Class ∼BreakException;
   temp0 := new ∼BreakException;
   (Var temp0).depth{∼BreakException} := Abzusteigende Schleifentiefe;
   Var temp0
}
```

Abbildung 7: Umsetzung des `break`-Statements

chend gesetzt wird, wie in Abbildung 7 zu sehen.

`continue` wird analog realisiert, indem eine ∼ContinueException geworfen wird.

Eine Schleife, die direkt ein `break` enthält oder die eine Unterschleife mit einem `break` mit entsprechender Tiefe enthält, wird mit einem TryCatch umschlossen, das die ∼BreakException fängt und prüft, ob die aktuelle Schleife die Zielschleife ist, oder eine umgebende. Im letzteren Fall wird das Attribut depth dekrementiert und die Exception erneut geworfen (siehe Listing 6).

```
while(Bedingung) {
    Rumpf;
}
//wird übersetzt als:
try {
    while(Bedingung) {
        Rumpf;
    }
} catch(~BreakException temp0) {
    if(temp0.depth > 1) {
        --temp0.depth;
        throw temp0;
    }
}
```

Listing 6: Anpassung der Schleife, wenn ein `break`-Statement vorkommt

Bei Schleifen mit `continue` wird innerhalb der Schleife das gleiche TryCatch benutzt (siehe Listing 7). Dabei muss sichergestellt werden, dass der Rumpf den Typ Void liefert, daher wird er mit einem Seq mit Unit als zweitem Unterausdruck umschlossen.

Da die Exceptions ∼BreakException und ∼ContinueException so realisiert werden, dass im Namen das Zeichen '∼' vorkommt, das in Java nicht für Klassennamen erlaubt ist, in Jinja hingegen schon, können sie im Benutzercode weder instantiiert noch direkt gefangen werden. Allerdings kann der Benutzer Throwable fangen. Damit dies nicht die Kontrollstrukturen beeinflussen kann, wird `catch` mit Throwable gesondert behandelt, siehe 4.12.

```
while(Bedingung) {
    Rumpf;
}
//wird übersetzt als:
while(Bedingung) {
    try {
        Rumpf;
    } catch(ContinueException temp0) {
        if(temp0.depth > 1) {
            --temp0.depth;
            throw temp0;
        }
    }
}
```

Listing 7: Anpassung der Schleife, wenn ein `continue`-Statement vorkommt

{temp0 : *Klassenname*;
temp0:=`New` *Klassenname*;
(`Var` temp0).*Konstruktorname*(*Argumente*);
`Var` temp0
}

Abbildung 8: Umsetzung einer Klasseninstantiierung in Jinja-Syntax

## 4.9 Klasseninstantiierung

Klasseninstantiierungen (`new`) werden in den New-Ausdruck von Jinja übersetzt, allerdings sind dabei einige zusätzliche Schritte notwendig: Da Jinja keine besondere Entsprechung für Konstruktoren besitzt, diese daher als gewöhnliche Methoden realisiert werden und folglich nicht automatisch von New aufgerufen werden können, werden diese durch einen entsprechenden Methodenaufruf explizit aufgerufen. Es muss eine temporäre Variable verwendet werden, um das erzeugte Objekt zwischenzuspeichern, da eine Konstruktormethode Void als Rückgabewert hat und nicht das Objekt. (siehe Abbildung 8)

Der Konstruktorname wird entsprechend der Namenskonvention eingesetzt (siehe 4.4.3). Eine Ausnahme stellt die Instantiierung der Klasse Object dar. In diesem Fall entfällt der Konstruktoraufruf, da für Object kein Konstruktor definiert wird.

Da eine Klasseninstantiierung auch in einem ExpressionStatement enthalten und damit wie ein Statement verwendet werden kann (siehe JLS [GJSB05, 14.8]), muss die Umsetzung in solch einem Fall dafür sorgen, dass der Rückgabetyp Void ist. Um dies zu erreichen, wird der Variablenzugriff auf die temporäre Variable am Ende des Blocks weggelassen.

## 4.10 Array-Erzeugung

Die Erzeugung von Array-Instanzen kann auf zwei verschiedene Arten erfolgen: Die Dimensionen werden übergeben oder es wird ein Initialisierer benutzt, der dann auch gleich Elemente in das Array einfügt. Der erste Fall bedarf der Auswertung der Dimensionsgrößen von links nach rechts und einer Überprüfung der Dimensionsgrößen auf Nicht-Negativität, bevor das Array tatsächlich erzeugt wird, wie in der JLS [GJSB05, 15.10.1] angegeben. Daher werden die Größen zunächst in lokalen Variablen zwischengespeichert, sodass die Ausdrücke nur einfach ausgewertet werden müssen. Darauf folgt die Überprüfung der Variableninhalte auf Nicht-Negativität.

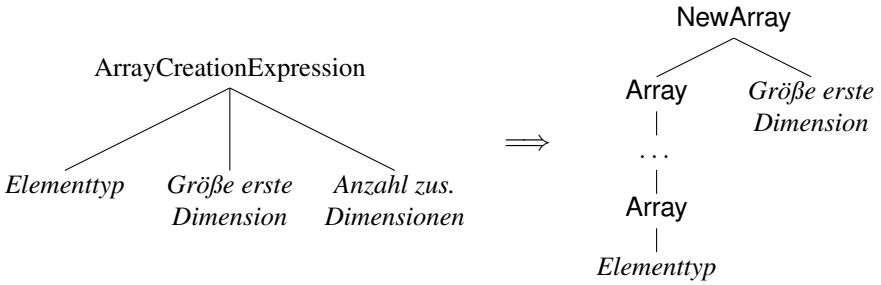
Da der NewArray-Ausdruck nur eindimensionale Arrays direkt erzeugen kann, werden mehrdimensionale Arrays realisiert, indem Arrays erzeugt werden, die als Komponententyp wiederum ein Array haben. Dazu wird für jede Dimension zunächst ein Array erzeugt, das die Größe dieser Dimension hat und dann für jedes enthaltene Array die nächste Dimension erzeugt:

```
Elementtyp[][][] a = new Elementtyp[m][n][o];
//wird übersetzt als:
int temp0 = m;
int temp1 = n;
int temp2 = o;

//Prüfung auf Nicht-Negativität
if(temp0 < 0 || temp1 < 0 || temp2 < 0)
    throw new NegativeArraySizeException();

Elementtyp[][][] a = new Elementtyp[temp0][][];
for(int temp3=0; temp3 < temp0; ++temp3) {
    a[temp3] = new Elementtyp[temp1][];
    for(int temp4=0; temp4 < temp1; ++temp4) {
        a[temp3][temp4] = new Elementtyp[temp2];
    }
}
```

Diese Schachtelung bricht ab, wenn keine weiteren Dimensionen mehr zu erzeugen sind, weil entweder der Komponententyp kein Arraytyp mehr ist oder weil für die weiteren Dimensionen keine Größe mehr angegeben ist. Die im Listing verwendete eindimensionale Arrayerzeugung wird umgewandelt, indem ein NewArray-Ausdruck erzeugt wird, bei dem die Anzahl der zusätzlichen Arraydimensionen, für die keine Größe angegeben ist, mit der Schachtelungstiefe des Komponententyps, also der Anzahl an Array-Typkonstruktoren, übereinstimmt.




Im zweiten Fall, wo bei der Array-Erzeugung ein Arrayinitialisierer angegeben ist, muss keine Prüfung auf Nichtnegativität stattfinden, da ein Initialisierer nur positive Elementanzahlen enthalten kann. Arrayinitialisierer können ineinander geschachtelt werden, um mehrere Dimensionen zu unterstützen, daher werden sie rekursiv verarbeitet: Für jeden Arrayinitialisierer wird ein Array erzeugt, dessen Größe gleich der Anzahl der Elemente des Initialisierers ist. Die Elemente dieses Arrays werden dann durch die Elemente des Initialisierers initialisiert, wobei für einen enthaltenen Arrayinitialisierer rekursiv die gleiche Prozedur angewendet wird. Dadurch werden diese Elemente, wie in der JLS gefordert [GJSB05, 10.6], in der Reihenfolge ausgewertet, wie sie im Quelltext vorkommen. Zur Veranschaulichung hier ein Beispiel:

```
int[][] a=new int[][]{{x, y}, {z}};
//wird übersetzt als:
int[][] a;
a = new int[2][];
a[0] = new int[2];
a[0][0] = x;
a[0][1] = y;
a[1] = new int[1];
a[1][0] = z;
```

## 4.11 Operatoren

### 4.11.1 Unäre Operatoren

Der unäre Operator '+' führt bei Java laut JLS [GJSB05, 15.15.3] lediglich zu einer Typumwandlung, wenn der Operand nicht ohnehin bereits vom Typ `int` ist. Da von Java2Jinja die Typen `int` und `char` identisch realisiert werden und der Operator nicht direkt auf andere unterstützte Typen anwendbar ist, da Java2Jinja kein Unboxing unterstützt, bewirkt der Operator keine Veränderung des Operanden, weswegen dieser unverändert weitergegeben wird.

Die Operatoren '$-$', '$\sim$' und '`!`' lassen sich durch Verknüpfung des Operanden mit einer Konstanten realisieren: '$-$' ist äquivalent zu einer Subtraktion des Operanden von 0 (siehe JLS [GJSB05, 15.15.4]). $\sim x$ ist laut JLS [GJSB05, 15.15.5] äquivalent zu $(-x) - 1$, wird aber hier als $(-1) - x$ umgesetzt, da $-1$ als Konstante keine zusätzliche Negationsoperation

```
int a = +x;
int b = -a;
int c = ~b;
boolean d = !y;
//wird übersetzt als:
int a = x;
int b = 0 - a;
int c = -1 - b;
boolean d = (y == false);
```

Listing 8: Umsetzung der unären Operatoren '+', '−', '∼' und '!'

*Variablenname* := `Var` *Variablenname* ± 1;
*Objekt.Feldname*{*Klasse*} := *Objekt.Feldname*{*Klasse*} ± 1;
*Array*[*Index*] := *Array*[*Index*] ± 1

Abbildung 9: Bei einer solchen Umsetzung wären Seiteneffekte in den Ausdrücken *Objekt*, *Array* und *Index* fatal

benötigt, im Gegensatz zu $-x$. Der '!'-Operator lässt sich auf viele Arten mit nur einer Operation umsetzen, hier wurde der Vergleich mit `false` gewählt. (siehe Listing 8)

Für die Operatoren '++' und '−−' ist ein größerer Aufwand erforderlich, da sie nicht nur sowohl als Präfix- und Postfixvariante existieren, sondern auch weil auf den Operand sowohl lesend als schreibend zugegriffen wird. Die gewöhnliche Lösung für das Problem, dass Werte mehrfach verwendet werden, aber ihre Ermittlung seiteneffektbehaftet sein kann, ist, den Wert in einer temporären Variable vorzuhalten. Für Schreibzugriffe hingegen muss ein Referenzzugriff möglich sein, was aber aufgrund des Fehlens von Referenzen auf beliebige Daten in Jinja wie in Java nicht über eine temporäre Variable möglich ist. Um dieses Problem zu lösen, wird ausgenutzt, dass nur ein Teil des Operandenausdrucks seiteneffektbehaftet sein kann, da ein solcher Ausdruck äußerlich stets ein Zugriff auf eine lokale Variable, ein Feld oder ein Arrayelement sein muss. Beim Zugriff auf eine lokale Variable ist kein Seiteneffekt möglich, daher müssen hier keine Vorkehrungen getroffen werden. Ein Feldzugriff enthält einen Ausdruck für die Klasseninstanz, welcher seiteneffektbehaftet sein kann und deshalb zwischengespeichert werden muss. Ein Arrayzugriff enthält einen Ausdruck für das Array, sowie für den Index des Arrayelements, so dass beides in temporären Variablen zwischengespeichert wird. So kann aus diesen drei Arten von Zugriffsausdrücken eine Variante erzeugt werden, die selbst keine Seiteneffekte mehr beinhalten kann, da diese auf temporäre Variablen ausgelagert sind. Zur Veranschaulichung zeigt Abbildung 9 die Problematik und die korrekte Umsetzung ist in Abbildung 10 zu sehen.

Der Unterschied zwischen Prä- und Postfixvariante der Operatoren ist, dass bei der Präfixvariante der neue Wert und bei der Postfixvariante der alte Wert des Operanden zurückgegeben wird. Um dies zu realisieren, wird bei der Präfixvariante der neue Wert zunächst berechnet, in eine temporäre Variable gespeichert, dann zugewiesen und am Ende der Wert aus der temporären Variable zurückgegeben. Bei der Postfixvariante wird der alte Wert in die tem-

```
{temp0 : Klasse; {temp1 : Arraytyp; {temp2 : Integer;
   temp0 := Objekt;
   temp1 := Array;
   temp2 := Index;
   Variablenname := Var Variablenname ± 1;
   (Var temp0).Feldname{Klasse} := (Var temp0).Feldname{Klasse} ± 1;
   (Var temp1)[Vartemp2] := (Var temp1)[Var temp2] ± 1
}}}
```

Abbildung 10: Bei dieser Umsetzung sind Seiteneffekte in den Ausdrücken *Objekt*, *Array* und *Index* tolerabel

```
{temp0 : Arraytyp; {temp1 : Integer;
   temp0 := Array;
   temp1 := Index;
   {temp2 : Integer;
      temp2 := (Var temp0)[Var temp1]%Wert;
      (Var temp0)[Var temp1] := Var temp2;
      Var temp2
}}}
```

Abbildung 11: Umsetzung für das Beispiel *Array*[*Index*]%=*Wert*

poräre Variable gespeichert, die Zuweisung findet mit dem berechneten Wert statt und der Variablenwert wird zurückgegeben. Sollte als Rückgabewert Void erwartet werden, sind beide Ausdrücke identisch, da dann die Umsetzung aus Abbildung 10 direkt angewendet wird.

### 4.11.2 Zuweisungsoperatoren

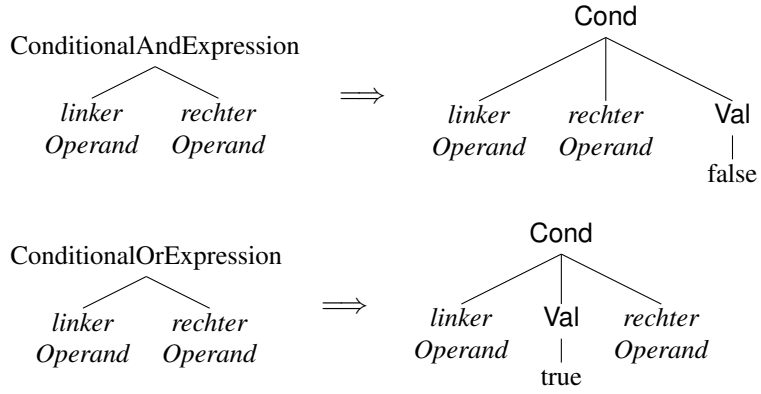
Dieser Abschnitt bezieht sich auf die Zuweisungsoperatoren '+=', '−=', '∗=', '/=', '%=', '&=', '|=', '^=', '<<=', '>>>=' und '>>=', der einfache Zuweisungsoperator '=' wird je nach linkem Operanden mit LAss, FAss oder AAss realisiert und hier nicht weiter erläutert.

Bei den Zuweisungsoperatoren besteht die gleiche Problematik wie bei den unären Operatoren '++' und '−−', also mögliche Seiteneffekte im linken Operanden, der lesend und schreibend verwendet wird. Die Lösung ist daher die gleiche, es werden also bei Feld- und Arrayzugriff Teile des Ausdrucks in temporären Variablen gehalten.

Der rechte Operand kann auch Seiteneffekte enthalten, deshalb ist auch hier die Anwendung einer temporären Variablen zum Zwischenspeichern des neuen Werts wie bei der Präfixvariante von '++' und '−−' erforderlich, da dieser nicht für die Zuweisung und für die Rückgabe mehrfach berechnet werden kann. In Abbildung 11 ist eine Beispielumsetzung zu sehen.

### 4.11.3 Binäre Operatoren

Jinja unterstützt die meisten binären Java-Operatoren direkt, deshalb wird hier nur auf die Fälle '`&&`' und '`||`' eingegangen. Diese werden mithilfe des Cond-Ausdrucks übersetzt, welcher hier in seiner funktionalen Rolle angewendet wird. Auf diese Weise wird auch die faule Auswertung korrekt unterstützt, da nur der Unterausdruck von Cond tatsächlich ausgewertet wird, der durch die Bedingung ausgewählt wird. In beiden Fällen wird der linke Operand als Bedingung für den Cond-Ausdruck verwendet, so dass dieser auf jeden Fall ausgewertet wird. Der rechte Operand bildet bei '`&&`' den Then-Ausdruck, da das Ergebnis nur dann wahr ist, wenn der linke Operand bereits `true` ergeben hat und der rechte ebenfalls `true` ergibt, entsprechend ist der Else-Ausdruck dann `false`. Bei '`||`' ist der Then-Ausdruck konstant `true`, da durch die Auswertung des linken Operanden dann bereits das Ergebnis feststeht, der Else-Ausdruck ist der rechte Operand, so dass bei unwahrem linken Operanden der rechte Operand das Ergebnis bildet.




### 4.11.4 Stringkonkatenation

Wenn mindestens einer der beiden Operanden einer Addition vom Typ java.lang.String ist, wird keine Addition, sondern eine Verkettung der Operanden durchgeführt. Dies geschieht jedoch nicht durch Erzeugung eines temporären String-Objekts für den Nicht-String-Operanden und anschließende Verkettung, wie in der JLS zunächst gefordert wird [GJSB05, 15.18.1.1], sondern indem ein Objekt vom Typ java.lang.StringBuilder verwendet wird, in Anlehnung an den Optimierungsvorschlag, der in der JLS folgt [GJSB05, 15.18.1.2]. Dadurch kann die Verkettung mit der Konvertierung kombiniert und so die Effizienz gesteigert werden, während gleichzeitig die Übersetzung einfacher wird, da z.B. der Wert `null` nicht besonders behandelt werden muss.

```
String s = Operand1 + Operand2;
//wird übersetzt als:
StringBuilder temp0 = new StringBuilder();
String s = temp0.append(Operand1).append(Operand2).toString();
```

## 4.12 Ausnahmebehandlung

Der TryCatch-Ausdruck ermöglicht es, gewöhnliche `try`-Statements von Java fast direkt in Jinja umzusetzen. Besondere Betrachtung erfordert jedoch die Möglichkeit, mehrere `catch`-Klauseln bei einem `try`-Statement zu haben, denn Exceptions, die in `catch`-Klauseln geworfen werden, können nicht von späteren Klauseln des selben `try` gefangen werden. Die naheliegende Umsetzung mehrerer `catch`-Klauseln durch das Schachteln von TryCatch-Ausdrücken würde jedoch zu genau dieser Möglichkeit führen. Das Problem wird hier dadurch behoben, dass jede `catch`-Klausel außer der äußersten mit einem weiteren TryCatch umgeben wird, welches sämtliche Exceptions fängt und zunächst in einer temporären Variable sichert. Nach dem Verlassen des gesamten `try`-Statements wird dann geprüft, ob eine Exception abgelegt wurde und diese entsprechend erneut geworfen. Es genügt zu prüfen, ob die temporäre Variable ungleich `null` ist, da `null` nie geworfen werden kann, sondern stattdessen eine NullPointerException tatsächlich geworfen wird, falls throw mit Argument `null` ausgeführt wird (siehe JLS [GJSB05, 14.18]).

```
try {
    try-Rumpf;
} catch (Typ e) {
    catch-Rumpf;
} catch (Typ2 e) {
    catch-Rumpf2;
}
//wird übersetzt als:
Throwable temp0 = null;
try {
    try {
        try-Rumpf;
    } catch (Typ e) {
        try {
            catch-Rumpf;
        } catch (Throwable temp1) {
            temp0 = temp1;
        }
    }
} catch (Typ2 e) {
    catch-Rumpf2;
}
if(temp0 != null) throw temp0;
```

Weiterhin muss der Sonderfall berücksichtigt werden, dass der Benutzer eine `catch`-Klausel für Throwable verwendet, da die Umsetzung von `return`, `break` und `continue` auch per Exception geschieht, dies aber für den Benutzer transparent sein muss. Es muss also sichergestellt sein, dass eine solche `catch`-Klausel für diese speziellen Exceptions nicht ausgeführt wird, sondern stattdessen die Exception nur weitergeworfen wird. Daher wird eine zusätzliche TryCatch-Schachtelungsebene für jede dieser speziellen Exceptions innerhalb des Throwable-TryCatch eingefügt, die ebenfalls den Mechanismus mit der oben genannten temporären Variable nutzt, um die Exception weiterzuwerfen. Diese eingefügten TryCatch-Ebenen ließen sich reduzieren, indem ausgewertet wird, ob tatsächlich innerhalb

ein entsprechendes Statement vorkommt, aber das wurde aus Zeitgründen nicht umgesetzt:

```
try {
    try-Rumpf;
} catch (Throwable t) {
    catch-Rumpf;
}
//wird übersetzt als:
Throwable temp0 = null;
try {
    try {
        try {
            try {
                try-Rumpf;
            } catch (ReturnException temp1) {
                temp0 = temp1;
            }
        } catch (BreakException temp2) {
            temp0 = temp2;
        }
    } catch (ContinueException temp3) {
        temp0 = temp3;
    }
} catch (Throwable t) {
    catch-Rumpf;
}
if(temp0 != null) throw temp0;
```

Die `finally`-Klausel ermöglicht es, einen Codeblock zu definieren, der sowohl bei normaler Ausführung als auch durch `throw`, `break`, `continue` oder `return` abrupt abgebrochener Ausführung beim Verlassen eines Blocks stets ausgeführt wird. Da die abrupt abgebrochene Ausführung von Java2Jinja durchgehend mithilfe von Exceptions realisiert wird, reduziert sich die Verarbeitung auf die zwei Fälle „Block normal verlassen“ und „Block mit Exception verlassen“. Der erste Fall wird realisiert, indem nach dem TryCatch der auszuführende Code abgelegt wird, während der zweite Fall durch ein zusätzliches TryCatch, welches den `finally`-Block ausführt und dann die Exception erneut wirft, umgesetzt wird. Dies führt zu Codeduplikation, die jedoch unkritisch ist, da nie beide Fälle gleichzeitig eintreten können und daher der Code stets nur einfach ausgeführt wird.

```
try {
    try-Rumpf;
} finally {
    finally-Rumpf;
}
//wird übersetzt als:
try {
    try-Rumpf;
} catch (Throwable temp0) {
    finally-Rumpf;
    throw temp0;
}
finally-Rumpf;
```

Die Typen von `try`-Block und `catch`-Block in einem TryCatch müssen gleich sein, siehe 3.2. Diese Bedingung wird dadurch eingehalten, dass von Java2Jinja für die `try`- und `catch`-Blöcke stets Ausdrücke generiert werden, die den Typ Void haben, indem bei den entsprechenden Sprachelementen Typanpassungen vorgenommen werden.

### 4.13 Zusicherungen

Mit einer Zusicherung (`assert`-Statement) lässt sich die Intention des Programmierers an geeigneten Programmstellen effektiver ausdrücken als mit Kommentaren, da Zusicherungen automatisch überprüft werden können. Allerdings führt diese Überprüfung zur Laufzeit zu einem Mehraufwand, weshalb diese deaktivierbar sein sollte. Das ist bei Java2Jinja auch durch eine globale Einstellung möglich, wie in 2.3 dargestellt, jedoch nicht klassenweise, wie es normalerweise laut JLS [GJSB05, 14.10] bei Java möglich ist. Wenn Zusicherungen deaktiviert sind, werden sie durch die Konstante Unit ersetzt, so dass sie weiterhin vom Typ Void sind, um dem Typsystem zu entsprechen.

Zusicherungen bestehen aus einem prüfbaren Ausdruck vom Typ `boolean` und optional einem zweiten Ausdruck, der beim Fehlschlag zusätzliche Informationen in der Fehlermeldung liefert. Umgesetzt werden fehlschlagende Zusicherungen durch das Werfen einer Exception vom Typ java.lang.AssertionError, wobei der Konstruktor der Exception die nötige Umwandlung des Arguments übernimmt, wenn dieses vorhanden sein sollte, während sonst der parameterlose Konstruktor verwendet wird:

```
assert Bedingung;
assert Bedingung2 : Argument;
//wird übersetzt als:
if(!Bedingung) throw new AssertionError();
if(!Bedingung2) throw new AssertionError(Argument);
```

## 5 Abhängigkeitsauflösung

Bei der Entwicklung war von Anfang an absehbar, dass Java2Jinja nicht den gesamten Java-Sprachumfang unterstützen wird. Um dennoch die Standardbibliothek mit möglichst wenigen Anpassungen verwenden zu können, werden nur die Klassen, Felder und Methoden tatsächlich umgewandelt, die benötigt werden, damit keine Abhängigkeiten fehlen.

Die Auflösung dieser Abhängigkeiten beginnt damit, dass eine Basismenge von Klassen, Feldern und Methoden gebildet wird. Diese ergibt sich aus den Klassen, die in einem wohlgeformten Jinjaprogramm immer enthalten sein müssen (siehe 3.4) und der Eingabedatei, die der Benutzer wählt. Aus dieser Eingabedatei werden alle Klassen, Methoden und Felder verwendet, also auch solche, die eigentlich nicht erreichbar wären, im Unterschied zur sonstigen Verarbeitung von Abhängigkeiten. Dies wurde so umgesetzt, damit auch Programme

ohne main-Methode übersetzt werden können und damit sich ganze Klassen auf Java2Jinja-Unterstützung testen lassen, ohne zusätzlichen Code einfügen zu müssen, der alle Elemente verwendet.

Von der Basismenge ausgehend wird die Übersetzung gestartet. Wenn dabei Abhängigkeiten auftreten, wird überprüft, ob die Auflösung für diese bereits angestoßen wurde, wobei unerheblich ist, ob sie bereits aufgelöst wurden oder noch darauf warten. Sollte dies nicht der Fall sein, wird die Auflösung für sie angestoßen. Dabei wird jedoch sichergestellt, dass die Klasse, die zu einem Feld oder einer Methode gehört, zuerst aufgelöst wird. Die Verarbeitung setzt sich rekursiv fort, bis alle aufgetretenen Abhängigkeiten verarbeitet wurden.

Für eine Klassenabhängigkeit wird bei der Verarbeitung eine leere Klasse erzeugt, für die nur Name und Oberklasse gesetzt werden, jedoch keine Felder und Methoden. Die Oberklasse ist bereits wiederum eine Abhängigkeit, daher wird für sie direkt die Auflösung angestoßen. Für eine Feldabhängigkeit wird ein Feld in der zugehörigen Klasse erzeugt, die deshalb auf jeden Fall bereits aufgelöst worden sein muss. Ein Feld enthält einen Namen, einen Typ und möglicherweise einen Initialisierer. Der Typ kann Abhängigkeiten enthalten, wenn es ein Klassentyp oder ein Arraytyp mit einer Klasse als Elementtyp ist, diese werden dann zur Auflösung angestoßen. Der Initialisierer wird als Ausdruck verarbeitet und kann entsprechend ebenfalls Abhängigkeiten enthalten, die wie bei Methodenrümpfen behandelt werden. Für eine Methodenabhängigkeit wird zur entsprechenden Klasse eine Methode eingefügt, die Name, Rückgabetyp, Parametertypen und in der Regel einen Rumpf hat. Daher wird für Rückgabe- und Parametertypen so wie beim Feldtyp verfahren. Der Rumpf wird als Anweisungsblock verarbeitet.

Bestimmte Anweisungen und Ausdrücke enthalten explizit oder implizit Abhängigkeiten. Explizite Klassenabhängigkeiten sind in lokalen Variablendeklarationen, Klasseninstantiierungen, `catch`-Klauseln, expliziten Typumwandlungen (`cast`) und `instanceof`-Ausdrücken gegeben. Explizite Methodenabhängigkeiten sind in Methodenaufrufen, expliziten Konstruktoraufrufen (mit `this` und `super`) und ebenfalls in Klasseninstantiierungen enthalten. Felder werden ausschließlich durch Feldzugriffe (lesend und schreibend) referenziert, daher sind dies die einzigen Quellen für Feldabhängigkeiten. Implizite Abhängigkeiten entstehen durch spezielle Umformungen durch Java2Jinja bei abstrakten Methoden, Stringkonkatenation, Arrayerzeugung (durch Prüfung negativer Dimensionsgrößen), Stringliterale, `assert`-Anweisungen und durch den automatischen Aufruf des Standardkonstruktors.

Die meisten Abhängigkeiten sind mithilfe der Namens- und Typauflösung des JDT direkt gegeben, jedoch sind zwei Sonderfälle zu betrachten: Java2Jinja generiert für jede Klasse einen Standardkonstruktor, wenn diese nicht selbst Konstruktoren definiert. Diese enthalten wiederum einen Aufruf des Konstruktors der Oberklasse, für den dann aber keine Abhängigkeitsinformationen vom JDT geliefert werden können, da diese Konstruktoren nicht im Java-Code vorhanden sind. Da aber die Oberklasse ohnehin als Abhängigkeit bereits verarbeitet wird, ist es kein Problem, den entsprechenden Konstruktor zu ermitteln. Der zweite Sonderfall entsteht durch die späte Bindung, die bei nicht-privaten Methoden in Java statt-

```
class A {
    void m() {}
}

class B extends A {
    void m() {}
}

class X {
    public static void main(String[] args) {
        A a = new B();
        a.m();
    }
}
```

Listing 9: Die Abhängigkeitsinformationen des JDT würden hier nur A.m() auflösen

findet. Das JDT liefert für einen Methodenaufruf nur die Methodenabhängigkeit für die Methode, die statisch aufgerufen werden würde (siehe Listing 9). Jedoch muss Java2Jinja auch Methoden auflösen, die diese Methode überschreiben, da zur Übersetzungszeit nicht bekannt ist, welche Methode bei einem Methodenaufruf tatsächlich erreicht wird.

Zu diesem Zweck werden bei jeder Methode, die aufgelöst wird, alle Unterklassen auf überschreibende Methoden überprüft. Jede gefundene überschreibende Methode wird als Abhängigkeit verwendet. Wenn eine Klasse aufgelöst wird, werden alle Oberklassen auf Methoden überprüft, die von einer der Methoden dieser Klasse überschrieben würden. Die gefundenen Methoden dieser Klasse werden ebenfalls als Abhängigkeiten verwendet.

Im Beispiellisting 9 würde also beim Auflösen der Methode A.m() die Unterklasse B durchsucht, falls diese bereits vorher aufgelöst wurde. Dabei würde die Methode B.m() als A.m() überschreibende Methode ermittelt und als Abhängigkeit verarbeitet. Wenn B erst aufgelöst würde, nachdem die Methode A.m() bereits verarbeitet wurde, würde die Auflösung dennoch funktionieren, da dann von Klasse B aus in der Oberklasse A die überschriebene Methode A.m() gefunden wird. Und somit wird ebenfalls B.m() als Abhängigkeit erkannt. Dieser beidseitige Ansatz stellt also sicher, dass die Reihenfolge der Auflösung nicht die Korrektheit beeinflusst.

# 6 Beispielprogramm

Das folgende Beispielprogramm benutzt einige der vorgestellten unterstützten Sprachelemente, um einen Eindruck von den Möglichkeiten des Plugins zu vermitteln. Java2Jinja erzeugt daraus erfolgreich eine ML-Datei, welche die Wohlgeformtheitsprüfung besteht, jedoch 4962 Zeilen umfasst und daher hier nicht gezeigt wird.

```
import java.util.Arrays;

abstract class Example {
    Object[] array = {};
    int value;
    public Example(int x) {
        value = x++;
        for(int i=0; i < value; ++i) {
            Object[] temp = new Object[array.length + 1];
            for(int j=0; j<array.length; ++j)
                temp[i] = array[i];
            switch(array.length % 8) {
            case 0:
                temp[array.length] = new Integer(array.length);
                break;
            case 1: case 2:
                temp[array.length] = new Boolean(beta(array));
                break;
            default:
                temp[array.length] = "" + temp.length;
            }
            array = temp;
        }
    }
    public abstract int alpha(int a, Object[] o);
    private boolean beta(Object[] tmp) {
        return (tmp.length & 3) == 1 || tmp.length > 0xA;
    }
}

class ExampleThread extends Thread {
    public static int count = 0;
    private Object obj;
    private String result;
    public ExampleThread(Object obj) {
        super("Thread");
        this.obj = obj;
    }
    public void run() {
        setName(getName() + "␣#" + increment());

        if(obj instanceof Integer) {
            int[] temp = new int[]{5, 3, 1};
            for(int i=0; i<((Integer)obj).intValue();)
                temp[++i % 3] <<= i;

            result = Arrays.toString(temp);
        }
        else
            result = obj instanceof String ?
                "\""+obj+"\"" : obj.toString();
    }
    private static synchronized int increment() {return count++;}
    public String toString() {return result;}
```

```
}

public class ExtensiveExample extends Example {
    public ExtensiveExample() {this(3);}
    private ExtensiveExample(int i) {
        super(i);

        alpha(value, array);
    }
    public static void main(String[] args) {
        if(args.length == 0) new ExtensiveExample();
        else new ExtensiveExample(args.length);
    }
    public int alpha(int a, Object[] objects) {
        ExampleThread[][] t = new ExampleThread[objects.length-a][a];
        for(int i=a, j=0; i < objects.length; ++i, j=0) {
            do {
                (t[i-a][j] = new ExampleThread(objects[i])).start();
            } while(++j < a);

            if(t[i-a][t[i-a].length - 1] == null) return j;
        }

        StringBuffer buffer = new StringBuffer();
        outer: for(int i=0; i < t.length; ++i)
            for(int j=0; j < t[i].length; ++j) {
                ExampleThread tt = t[i][j];
                try {
                    if(tt == null) continue;
                    tt.join();
                    buffer.append(tt).append("\n");
                } catch (InterruptedException e) {
                    buffer.append('!').append(tt.getName());
                    break outer;
                } finally {
                    tt.setName("Finished␣" + tt.getName());
                }
            }

        return 0;
    }
    private boolean beta() {
        assert false : "This␣method␣is␣not␣meant␣to␣be␣called";
        return false;
    }
}
```

# 7 Ausblick

Die Studienarbeit hat gezeigt, dass eine maschinelle Übersetzung von Java-Code in die Syntax von Jinja für einen wesentlichen Teil des Java-Sprachumfangs möglich ist. Dieses Kapitel soll zum Abschluss Hinweise darauf geben, welche weiteren Sprachelemente sich mit den bestehenden Elementen von Jinja in Zukunft übersetzen lassen könnten und welche Erweiterungen der Jinja-Syntax sinnvoll wären, um noch zusätzliche Sprachelemente zu unterstützen oder bestehende besser zu übersetzen. Dabei ist insbesondere zu beachten, dass die Einfachheit der Transformation gewahrt bleibt, damit die Gefahr von fehlerhaften Übersetzungen gering ist.

## 7.1 Bisher nicht unterstützte Sprachelemente

Die primitiven Ganzzahltypen **`byte`** und **`short`** könnten analog zu `char` durch den Typ Integer dargestellt werden und müssten bei Typumwandlungen entsprechend behandelt werden.

**Aufzählungen**(`enum`) können durch Klassen dargestellt werden, die automatisch mit einer Instanz initialisierte Felder für jede Konstante enthalten und deren Konstruktor nicht vom Benutzer aufgerufen werden kann, sodass nur diese vorgegebenen Instanzen der Klasse existieren können, damit die Verwendung des gewöhnlichen Gleichheitsoperators möglich ist. Die Verwendung von `enum`-Konstanten ohne Angabe der Klasse in `switch`-Fällen kann mithilfe des JDT zu gewöhnlichen Feldzugriffen aufgelöst werden.

**Ineinander geschachtelte Klassen** lassen sich zunächst leicht durch eine Namenskonvention umsetzen, was aber nur genügt, wenn sie auch als `static` deklariert sind. Nichtstatische innere Klassen erfordern zusätzlich eine Referenz auf die zugehörige Instanz der äußeren Klasse um auf deren Felder zugreifen zu können. Für innere Klassen innerhalb eines Methodenrumpfes müssen die verwendeten lokalen Variablen als `final` deklariert sein, weshalb diese Variablen dann bei der Instantiierung in das erzeugte Objekt kopiert werden können und die Zugriffe auf diese entsprechend als Feldzugriffe umgesetzt werden. Anonyme Klassen lassen sich genauso realisieren, indem sie mit einem eindeutigen Namen versehen werden und ein Konstruktor gemäß JLS [GJSB05, 15.9.5.1] generiert wird.

Durch eine aufwändige Umformung ließen sich **Interfaces** unterstützen, indem sie als Klassen realisiert werden, die eine Objektreferenz der Klassen speichern können, die das Interface implementieren. Methoden des Interfaces müssten dann anhand des Typs der enthaltenen Referenz den Aufruf an die eigentliche Methode weiterleiten. Eine Vererbungshierarchie von Interfaces ließe sich unterstützen, indem Oberinterfaces auch ihre Unterinterfaces als Referenzen enthalten können und Methodenaufrufe entsprechend durch mehrere Stufen von Interfaceobjekten geleitet werden. Eine Umwandlung einer Interfaceinstanz zurück in eine Klasseninstanz würde durch mehrere Typumwandlungsstufen realisiert. Letztlich läuft eine solch aufwändige Umwandlung aber dem Ziel der Einfachheit entgegen und sollte da-

her gut überlegt sein.

Da Feldinitialisierer für statische Felder bereits unterstützt werden und somit ein Ort für die statische Initialisierung bereits besteht, ließen sich auch **statische Initialisierer**, die einen Block ausführen, wenn eine Klasse initialisiert wird, implementieren, indem diese in der richtigen Reihenfolge zwischen den statischen Feldinitialisierern eingefügt werden.

Das **erweiterte `for`-Statement** („foreach“) lässt sich ohne weiteres nur für Arrays umsetzen, da für Collection-Klassen die Iteration durch java.lang.Iterable realisiert wird, was jedoch Generics und Interfaces erfordert. Für Arrays lässt sich anhand der JLS [GJSB05, 14.14.2] die entsprechende `for`-Schleife bilden, welche dann wie eine gewöhnliche `for`-Schleife konvertiert wird.

Obwohl Jinja bei jedem Feldzugriff die Angabe der deklarierenden Klasse erfordert, kann der **`super`-Feldzugriff** nicht mit der bisherigen Feldrepräsentation realisiert werden, da gleichnamige Felder in einer Klassenhierarchie sich verdecken. Es müsste eine ähnliche Namenskonvention wie bei Methoden eingeführt werden, um auch gleichnamige Felder in Oberklassen direkt ansprechen zu können.

Für die von Jinja unterstützten Primitiv-Typen ließe sich **automatisches Boxing/Unboxing** durch Umsetzung in explizite Objektinstantiierungen/Methodenaufrufe leicht implementieren, da der Parser des JDT bereits die Information bereitstellt, ob ein Ausdruck eine Anwendung von Boxing oder Unboxing impliziert.

## 7.2 Vorschläge für Jinja-Erweiterungen

Um die primitiven Datentypen `long`, `float` und `double` in Jinja-Syntax umsetzen zu können, wären **zusätzliche Typen mit größerem Wertebereich bzw. Gleitkommadarstellung** notwendig. Diese durch mehrere Variablen vom Typ Integer darzustellen wäre bei `long` bereits schwierig, die Gleitkommaarithmetik jedoch durch Ganzzahloperationen zu realisieren wäre sehr aufwändig, sodass eine solche Umsetzung kaum in Frage kommt.

Statt der im vorherigen Abschnitt angedeuteten aufwändigen Umwandlung von Interfaces in Klassen, würde eine **direkte Unterstützung für Interfaces** zu wesentlich kompakterer Darstellung führen, die Übersetzung wieder auf die gewünschte Einfachheit bringen und auch mehr Informationen für die Weiterverarbeitung bereithalten.

Um unter anderem die Collection-Klassen nutzbar zu machen, wäre es hilfreich, eine **native Unterstützung für generische Klassen und Methoden** zu haben. Zwar lassen sich diese auch durch Bildung der „Erasure“ (siehe JLS [GJSB05, 4.6]) ähnlich der Umsetzung in der Java Virtual Machine realisieren, jedoch würden dadurch viele Informationen entfallen, die im Typsystem enthalten sind und bei der Weiterverarbeitung sinnvoll verwendet werden könnten.

Die Darstellung statischer Felder und Methoden durch Weiterreichen eines globaleinheit-

lichen Objekts verursacht Probleme mit den Klassen, deren Struktur durch Jinja teilweise festgelegt ist, beispielsweise die Klasse java.lang.Object. Da sich zu dieser Klasse kein Feld hinzufügen lässt, welches das globale Objekt enthält, können in der internen Implementierung der Methoden von java.lang.Object keine Objektinstanzen erzeugt werden, da bei diesen Instantiierungen auch das Objekt weitergereicht werden müsste. Eine Erweiterung der Jinja-Syntax, die **nativ statische Felder und Methoden** unterstützt, würde die Notwendigkeit für dieses Feld entfallen lassen und diese Problematik damit entschärfen. Auch würden einige Sonderfälle entfallen und so die Übersetzung wieder einfacher werden.

Kontrollflussverändernde Statements (`return`, `break` und `continue`) werden bisher durch eine Zweckentfremdung der Ausnahmebehandlung umgesetzt, was jedoch im Sinne der eigentlichen Semantik nicht wünschenswert ist. Eine direkte Unterstützung dieser Sprachelemente oder die Möglichkeit, **Sprunganweisungen** durch Jinja-Syntax auszudrücken, würden eine elegantere Repräsentation ermöglichen.

Um einen größeren Teil der Standardbibliothek nutzen zu können, wären **weitere nativ umgesetzte Methoden**, beispielsweise für die Standardausgabe oder den Zugriff auf das Dateisystem, hilfreich. Allerdings würde dies unter Umständen über den eigentlichen Sinn der Umwandlung nach Jinja hinausgehen, da sich die entsprechenden nativen Methoden auch für die Wohlgeformheitsprüfung und weitergehende Analysen verwenden lassen, wenn sie lediglich als Stubs implementiert sind.

# Literatur


[BJ66] BÖHM, CORRADO und GIUSEPPE JACOPINI: *Flow diagrams, turing machines and languages with only two formation rules*. Commun. ACM, 9(5):366–371, 1966.

[GJSB05] GOSLING, JAMES, BILL JOY, GUY STEELE und GILAD BRACHA: *Java(TM) Language Specification, The (3rd Edition)*. Addison-Wesley Professional, 2005.

[KN06] KLEIN, GERWIN und TOBIAS NIPKOW: *A Machine-Checked Model for a Java-Like Language, Virtual Machine and Compiler*. ACM Transactions on Programming Languages and Systems, 28(4):619–695, 2006.

[Loc08] LOCHBIHLER, ANDREAS: *Type Safe Nondeterminism - A Formal Semantics of Java Threads*. In: *International Workshop on Foundations of Object-Oriented Languages (FOOL 2008)*, January 2008.

[LY99] LINDHOLM, TIM und FRANK YELLIN: *Java Virtual Machine Specification*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.

## Eidesstattliche Erklärung

Hiermit versichere ich, die vorliegende Arbeit selbständig und unter ausschließlicher Verwendung der angegebenen Literatur und Hilfsmittel erstellt zu haben.

Die Arbeit wurde bisher in gleicher oder ähnlicher Form keiner anderen Prüfungsbehörde vorgelegt und auch nicht veröffentlicht.

Karlsruhe, den 11. März 2010

JONAS THEDERING