STUDIENARBEIT

Berufsakademie Stuttgart - Außenstelle Horb a. N.
Staatliche Studienakademie

Fachrichtung Informationstechnik

Thema

Weiterentwicklung der Java-Codegenerierung zur Ausführung von parametrisierten Datentypen

Juni 2004

Eingereicht von:

Markus Haas
Schonachbach 14

78098 Triberg

Firma:

GFT Technologies AG
Leopoldstraße 1

78112 St. Georgen

Betreuer:
Felix Reichenbach

Zusammenfassung der Arbeit

Aufgabenstellung

Aufgabe ist es einen bestehenden Java-Compiler so zu erweitern, dass Programme aus der Studienarbeit "`Typinferenz in Java`" ausführbar werden.
Folgende Anforderungen werden an das System gestellt:

- In Abstimmung mit der Studienarbeit "`Typinferenz in Java`" soll der abstrakte Syntax für getypte Java-Programme festgelegt werden.
- Die Klassen der abstrakten Syntax sind mit Methoden zu versehen bzw. zu ergänzen, die JVM Code erzeugen.
- Das Hauptproblem, das zu klären ist, ist ob die Codeerzeugung für Attribute, Variablen und Methoden mit Typtermen als Typen genauso zu implementieren ist, wie die Typen im Original-Java.

Kurzbeschreibung

Die vorliegenden Studienarbeit beschäftigt sich mit der Aufgabe die Programmiersprache Java so zu erweitern, dass eine Typisierung von Attributen bei Klassen möglich ist. Dazu soll ein bestehender Java Compiler erweitert werden. Zentraler Punkt bildet die Codeerzeugung, da durch eine vorige Studienarbeit bereits die Aufgaben Erweiterung der lexikalischen Regeln, Erweiterung der Grammatik und Aufbau der semantischen Analyse durchgeführt wurden. Ziel ist es, dass der Compiler ein gültiges Classfile erzeugt. Des weiteren soll dies auch für Programme, bei denen der Compiler die Typen berechnet (→ Studienarbeit: "`Typinferenz in Java`"), gelingen.

Zielsetzung

Ziel der Studienarbeit ist es obiger Aufgabenstellung gerecht zu werden und mit dem Compiler gültigen Java Bytecode für parametrisierte Klassen zu generieren.

Umfeld

Die Entwicklung und das Debugging des Compiler erfolgte mit Eclipse. Als Versionsverwaltungssoftware kam CVS zu Einsatz. Zum erstellen des Compilers wurde ein Linux Derivat benutzt um "`jay`" und "`make`" nutzen zu können, des weiteren wurde "`JLex`" eingesetzt. Um die vom Compiler erzeugten Classfiles analysieren zu können wurden die Disassembler "`jvmDisassembler`", "`jad`" und "`ClassCracker`" benutzt sowie ein Hex Editor.

Ergebnis

Die Aufgabenstellung sah die Codeerzeugung von parametrisierten Klassen vor. Dies wurde zu einem Teil verwirklicht. So erzeugt der Compiler jetzt korrekten Bytecode zu Use Cases, die parametrisierte Klassen enthalten.

Ehrenwörtliche Erklärung

Erklärung: Ich habe die Studienarbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt.

Ort, Datum und Unterschrift des Studierenden

Inhaltsverzeichnis

1 Erklärung von Abkürzungen

ADT	Abstract Data Type
ANSI	American National Standards Institute
CVS	Concurrent Versioning System
EJB	Enterprise Java Beans
IDE	Integrated Development Environment
ISO	International Standards Organization
J2EE	Java 2 Plattform Enterprise Edition
J2ME	Java 2 Micro Edition
J2SE	Java 2 Standard Edition
JBC	Java Bytecode
JC	Java Compiler
JDBC	Java Database Connectivity
JDK	Java Development Kid
JIT	Just In Time
JRE	Java Runtime Environment
JSP	Java Server Page
JVM	Java Virtual Machine
JPEG	File Interchange Format
JPG	File Interchange Format
MVC	Model View Controller
PDA	Personal Digital Assistants
PDF	Portable Dokument Format
SGML	Structured Generalized Markup Language
SVG	Scalable Vector Graphics
UML	Unified Modeling Language
VM	Virtual Machine
W3C	World Wide Web Consortium
WAR	Web Application Archive
XML	Extensible Markup Language

2 Problemstellung

2.1 Aufgabe

In dem vorliegendem Java Compiler, wurden bereits im Vorfeld durch eine andere Studienarbeit [REI03] die lexikalischen Regeln, der Parser und die semantische Analyse erweitert (siehe Kapitel 2.3). Nun soll in einem weiteren Schritt die Codeerzeugung für parametrisierte Klassen implementiert werden.

2.2 Motivation

Warum eine Erweiterung des Compilers bzw. des Java Sprachumfangs? Die Antwort hierauf ist der Vorteil parametrisierbarer Klassen. Parametrisierbare Klassen geben dem Programmierer die Möglichkeit Klassen zu implementieren, die auf Daten arbeiten ohne ihren Typ zu kennen (ähnlich dem Template Mechanismus in C++). Dies ist vor allem bei der Implementierung von Listen, Hashtabellen und ähnlichen Container Klassen zur Speicherung von Objekten interessant. Sun hat die Einführung parametrisierbarer Klassen in ihrem neuem JDK 1.5 angekündigt, welches in einer Beta Version zur Verfügung steht. Ein Beispiel einer parametrisierbaren Klasse zeigt Listing 1.

```
01    class StoreSomething<A>
02    {
03          A something;
04
05          StoreSomething(A someth)
06          {
07                something = someth;
08          }
09
10          void set(A someth)
11          {
12                something = someth;
13          }
14
15          A get()
16          {
17                return something;
18          }
19    }
20
21    class MainStoreSomething
22    {
23          public static void main(String[] args)
24          {
25                StoreSomething<String> var;
26                var = new StoreSomething("I'm a string!");
27                System.out.println(var.get());
28          }
29    }
```

Listing 1: Beispiel für eine parametrisierte Klasse

Die Klasse StoreSomething dient dazu ein Objekt beliebigen Typs zu speichern, dazu enthält sie den Typparameter A. Außerdem beinhaltet die Klasse einen Konstruktor sowie eine get- und set- Methode zum manipulieren des Attributs "`something`". Wenn die Klasse deklariert wird, ist es nötig den Typ der gespeichert werden soll, zu übergeben. Dies geschieht in Zeile 25, wenn "`<String>`" auf den Klassenname der parametrisierbaren Klasse folgt. In Zeile 26 wird dann bei der Instanzierung dem Konstuktor ein String übergeben, der dann in dem Attribut gespeichert und später ausgeben wird.

2.3 Geschichte des Projekts

Im Rahmen der Vorlesungen Informatik und Software Engineering entstand der Compiler durch eine Gruppe des Studienjahrgangs IT2000. Im dritten Semester wurde dazu ein Entwurf für das Klassendesign erarbeitet (Ende 2001). Anfang des nächsten Jahres erfolgte (im nächsten Semester) die Implementierung. Felix Reichenbach, bekam im fünften und sechsten Semester im Kontext einer Studienarbeit, die Aufgabe die semantische Analyse dahingehen zu erweitern, dass eine Typisierung von Attributen bei Klassen möglich wird. Um dieses Ziel zu erreichen wurde der Parser sowie die semantische Analyse erweitert.
Gegen Ende des Jahres 2003 wurde der Compiler (auch im Rahmen einer Studienarbeit) an Thomas Ott und Markus Haas übergeben. Herrn Ott's Aufgabe war es die Typen von dem Compiler berechnen zu lassen [OTT04]. Die Aufgabe von Herr Haas war die Fortsetzung der Studienarbeit von Herr Reichenbach. Programme, die durch seine Arbeit spezifiziert sind, sollen auch in Abstimmung mit der Arbeit von Herr Ott ausführbar werden bzw. gültigen Bytecode liefern.

3 Analyse des Problems

3.1 Status Ermittlung bei Projektbeginn

In diesem Abschnitt soll dargelegt werden in welchem Zustand der Compiler bei der Übergabe war. Aus dieser Analyse lies sich das weitere Vorgehen planen (siehe Kapitel 3.2).
Der Compiler wurde in Form einer gepackten Datei übergeben. Außerdem wurde die Studienarbeit des Herrn Reichenbach sowie die Daten zum Innovator Repository überreicht[1]. Das ursprüngliche und vollständige Innovator Repository ist bei einem Versionswechsel zerstört worden, bzw. die Konvertierung auf eine neue Server Version war nicht möglich, deshalb beinhaltete das übergebene Repository nur die wichtigsten Klassen mit Attributen und Methoden. Der Compiler lies sich nach dem entpacken fehlerfrei über das Makefile bauen und compilierte einige Testklassen zu gültigem Bytecode. Allerdings fiel auf, das einige Testklassen im gleichen Verzeichnis wie die Compilerklassen lagen. Außerdem entsprachen manche Klassennamen nicht den allgemeinen Java - Konventionen.

[1] /bahome/projekt/inoprj/Compiler1_8_1.ir; Benutzername: ADMIN; Passwort: <leer>

3.2 Planung des weiteren Projektverlaufs

Aufgrund der vorherigen Analysen wurde folgendes Vorgehen für den weiteren Projektverlauf beschlossen. Im weiteren wird auf die einzelnen Punkte eingegangen. Dabei wird festgelegt welche Aufgaben in den einzelnen Bereichen erfüllt werden sollen. Im Kapitel 3.2.10 ist dann ein Verweis auf die tatsächliche Durchführung zu finden.

- Anlegen eines CVS Repositories [HEL04]
- Erweiterung des Innovator Repositories [MID04]
- Umbenennung einiger Klassen
- Bildung von Packages und erstellen von Unterverzeichnissen
- Startbefehl des Compilers ändern / einstellbarer Debug Level
- Definition von Use Cases
- Funktionsanalyse des Compilers
- Eventuelle Bugfix oder Erweiterungen
- Implementierung der Codeerzeugung

3.2.1 Anlegen eines CVS Repositories

Der Hauptgrund, der für die Einrichtung eines CVS Repositories sprach war, dass zwei Studienarbeiten parallel mit einem Projekt an dem Compiler involviert waren. Dies bringt zwangsläufig das Problem mit sicht, dass mehrere Personen gleichzeitig zwei gleiche Dateien editieren. Mit dem Open Source Softwarewerkzeug CVS hält man zum einen alle Dateien eines Projekts an einer zentralen Stelle, zum anderen übernimmt CVS die Versionierung. Dies ermöglicht einem Benutzer jederzeit, eine vorige Version einer Datei oder des gesamten Projekts anzusehen bzw, zurück zu spielen. Außerdem können sich mehrere Benutzer eine lokale Kopie aus dem Verzeichnisbaum auschecken, dann unabhängig voneinander bearbeiten und wieder in das Repository einchecken. CVS nutzt dabei einen 'merge'-Ansatz, d.h. wenn Konflikte auftreten sind diese von bzw. unter den Entwicklern zu lösen. Eine Datei wird erst dann in das Repository geschrieben, wenn alle Konflikte aufgelöst wurden.

3.2.2 Erweiterung des Innovator Repositories

Da ein vollständiges Klassendiagramm die Einarbeitung sowie den Überblick über ein solches Projekt stark vereinfacht, sollte in diesem Schritt das vorhandene Innovator Projekt mit dem Sourcecode verglichen und aktualisiert bzw. ergänzt werden. Außerdem sollte auf diese Art eine gemeinsame Basis als Ausgangspunkt in Abstimmung mit der Studienarbeit “`Typinferenz in Java`“ [OTT04] geschaffen werden. Dies beinhaltete außerdem die gemeinsame Festlegung der abstrakten Syntax für die getyten Java Programme.

3.2.3 Umbenennung einiger Klassen

Auch dieser Schritt sollte die Übersicht über das gesamte Projekt erleichtern. So sollte aufgrund einer besseren Lesbarkeit des Sourcecodes sowie aufgrund allgemeiner Konventionen (Sun Namenskonvention) einige Klassen umbenannt werden. Außerdem sollten in dieser Phase fehlende Modifiers bei Klassen, Attributen und Methoden ergänzt werden.

3.2.4 Bildung von Packages und erstellen von Unterverzeichnissen

Der Compiler wurde wie bereits erwähnt in Form einer gepackten Datei übergeben. Beim Entpacken wurden alle Dateien des Compilers (Sourcecode und Testklassen) ohne Unterverzeichnisse abgelegt. Hier sollten die Testklassen als Package aus dem Root-Verzeichnis der Compiler-Klassen in ein Unterverzeichnis verschoben werden. Außerdem sollen weitere Verzeichnisse angelegt werden:

- doc — Javadoc des Compilers
- hama — Dateien die diese Studienarbeit betreffen
- otth — Dateien die die Studienarbeit "`Typinferenz in Java`" [OTT04] betreffen
- Rest — restliche allgemeine Dateien (siehe Kapitel 4.4.4 und 4.4.5)
- test — das angesprochene Package mit Testklassen, die bei der Übergabe vorhanden waren

3.2.5 Startart des Compilers ändern / einstellbarer Debug Level

Bei der Übergabe wurde der Compiler wie folgt aufgerufen:

```
java MyCompiler < <File>
```

Da dieser Aufruf im Umgang mit Eclipse Schwierigkeiten machte und es außerdem wünschenswert wäre, optional einen Level für die Anzahl der Debug Ausgaben angeben zu können soll der Compileraufruf in folgenden geändert werden:

```
java MyCompiler <File> [Debug Level]
```

Wobei wenn kein Debug Level angegeben wurde auch keine Debug Ausgaben angezeigt werden sollen. Welche Debug Ausgaben wie eingeschaltet werden erfährt man mit dem Aufruf:

```
java MyCompiler
```

Genauere Informationen sind im Kapitel 4.4.2 zu finden.

3.2.6 Definition von Use Cases

In dieser Phase sollen einige Use Cases definiert werden. Diese Use Cases sollen möglichst den Java Sprachumfang abdecken, damit mit ihnen eine intensive Testphase abgestoßen werden kann (nächster Punkt). Des weitern sollen auch Use Cases erstellt werden, die typisierte Klassen abbilden.

3.2.7 Funktionsanalyse des Compilers

Die Funktionsanalyse des Compilers bildet die Grundlage des weiteren Vorgehens. Hier soll aufgezeigt werden welche Use Cases der Compiler richtig compiliert und welche nicht. Durch eine Modifizierung der Use Cases soll dann aufgeklärt werden, welche Java Befehle der Compiler unterstützt bzw. nicht unterstützt, oder in welchem Kontext eventuelle Fehler bei der Compilation auftreten (Kapitel 5.1).

3.2.8 Eventuelle Bugfix oder Erweiterungen

Falls im Bereich der Funktionsanalyse des Compiler irgendwelche Bugs bei der Codegenerierung entdeckt werden sollten, dann ist diese Phase dafür vorgesehen diese Bugs zu fixen (sofern sie für den weiteren Verlauf relevant sind). Außerdem ist es denkbar, dass im Bereich der Funktionsanalyse festgestellt wird, dass eine Funktion, die im weiteren Verlauf benötigt wird, noch nicht implementiert ist. An dieser Stelle soll dann eine Implementierung erfolgen (näheres siehe Kapitel 5.2).

3.2.9 Implementierung der Codeerzeugung

Schlussendlich soll die Codeerzeugung für parametrisierte Klassen implementiert werden (Kapitel 5.3).

3.2.10 Tatsächliche Durchführung

Die hier erwähnten Punkte wurden nach der Planungsphase wie zuvor beschrieben umgesetzt. Genauere Informationen hauptsächlich zu den Punkten 3.2.7-3.2.9 sind Kapitel 5 “`Umsetzung`“ zu entnehmen.

4 Grundlagen

4.1 Java

“`Java`“ [SUN04a] ist eine Programmiersprache, die ursprünglich von der Firma Sun Microsystems entwickelt wurde. Der offizielle Geburtstag ist der 23. Mai 1995. Zwar begann die Geschichte von Java schon viel früher, aber an diesem Tag verkündete der Netscape Gründer Marc Andreessen, dass die im Dezember 1995 auf den Markt kommende Netscape Navigator Version 2.0, die Java Technologie von Sun beinhalten und somit einen breiten Publikum zur Verfügung stehen wird.

Java ist einfach und klein. Java ist für Einsteiger und Profis gleichermaßen geeignet. Einfache Fenster-Anwendungen, Browser Applets oder komplexe Client-, Server- und Datenbank-Module lassen sich dank der klaren Spracheigenschaften schnell umsetzen. Java kann in der "`Java 2 Micro Edition`" [SUN04d] auf sehr kleinen (mobilen) Geräten mit geringen Ressourcen sowie geringer Prozessorleistung und Speicherplatz laufen. Die "`Java 2 Standard Edition`" [SUN04b] ist die Desktop Variante für Fenster Applikation. Für mehrschichtige Client-Server-Anwendungen wird die "`Java 2 Enterprise Edition`" [SUN04c] angeboten. Hierin finden sich auch "`Java Servlets`" [SUN04e] und "`Java Server Pages`" [SUN04e] für die Erstellung interaktiver Webseiten.

Java ist objektorientiert und verteilt. Wie der Name schon sagt, konzentriert man sich bei der objektorientierten Programmierung auf die Erstellung von Objekten. Objekte können aus Daten (Attributen) und Funktionen (Methoden) bestehen, Methoden dienen dazu die Daten zu manipulieren. Im Idealfall erzeugt man mit objektorientierter Programmierung wiederverwendbare Objekte. Bei fortschreitender Programmierung kann man dann auf einen immer größer werdenden Pool an schon bestehenden Objekten mit Funktionalitäten zurückgreifen. Auf entfernt liegende Objekte können Java Programme einfach über das Internet zugreifen.

Java ist compiliert und schnell. Java Programme werden im Gegensatz zu vielen anderen Compiler Sprachen nicht direkt in eine für den Prozessor verständliche Menge an Befehlen übersetzt, sondern zunächst in den sogenannten Java Bytecode (JBC) umgewandelt. Dieser Bytecode wird bei der Compilation einer Klasse oder Interfaces in das Java Classfile geschrieben. Auf das Format des Classfiles wird in Kapitel 4.2 eingegangen. Der JBC ist sehr maschinennah. Erst auf dem Zielrechner und bei der Ausführung wird der Bytecode im Java Runtime Environment (JRE) der JVM (Java Virtual Machine) ausgeführt. Dabei abstrahiert die JVM die tatsächliche Hardware und das Betriebssystem, des Rechners der das Java Programm ausführt. Dazu werden aus allen kompatiblen Rechnerplattformen diejenigen Funktionen, die jede dieser Plattformen ausführen kann extrahiert. Für die Ausführung eines Java Programms wird der jeweilige ausführende Rechner unter einer standardisierten Oberfläche 'versteckt', die dem Java Programm einen Rechner 'vorgaukelt'. Startet man also ein Java Programm, so wird zuerst ein Programm geladen, das die zur Verfügung stehenden Systemressourcen soweit abstrahiert, dass sie den Standards der VM entsprechen. Diese Virtual Machine gibt es für jede Plattform auf der Java läuft. Der Nachteil von Java liegt auf der Hand: Da das Programm nicht betriebssystemspezifisch compiliert worden ist, muss der Bytecode von der VM interpretiert werden, und das dauert recht lange. Aus diesem Grund sind 'klassich' compilierte Programme immer noch etwas schneller. Der Vorsprung schwindet allerdings, da die Entwicklung der Virtual Machines fortschreitet, und die so genannten JIT Compiler (Just In Time) in der Lage sind, Java Programme nicht anfangs am Stück zu interpretieren, sondern immer dann, wenn das entsprechende Programmsegment zur Ausführung kommt (Just In Time). Abbildung 1 zeigt den schematischen Ablauf der Compilation bzw. der Ausführung. Außerdem werden beim compilieren eines Java Programms alle abhängigen Klassen benötigt, diese werden in den durch den CLASSPATH angegebenen Pfaden gesucht.

Java ist robust und sicher. Die Kombination Compiler und Laufzeitumgebung hat für Java unter einem anderen Gesichtspunkt große Vorteile. Einerseits kann der Compiler auf Typenfehler und dergleichen prüfen, anderseits kann die JRE Fehler während der Laufzeit abfangen und behandeln. Außerdem überwacht die VM zur Laufzeit die Ausführung des Programms und verhindert so z.B., dass ein Programm in fremde Speicherbereiche oder über Arraygrenzen hinweg liest oder schreibt.

Java ist plattformunabhängig und portierbar. Der Compiler übersetzt das Programm in den Bytecode, der völlig unabhängig von irgendeinem bestehenden Prozessor und Betriebssystem ist. Dieser Bytecode kann als Zwischensprache zwischen der Hochsprache Java und dem Maschinencode gesehen werden. Außerdem ist der Bytecode für die Plattformunabhänigkeit verantwortlich (siehe Abbildung 1). Der Richtigkeit halber müsste man allerdings sagen, dass Java Programme nur auf einer einzigen Plattform laufen: der Java Virtual Machine. Dies ist der Rechner, der mittels des gleichnamigen Programms simuliert wird. Die VM gibt es für jede Plattform auf der Java läuft. Sie selbst ist natürlich nicht plattformunabhängig.

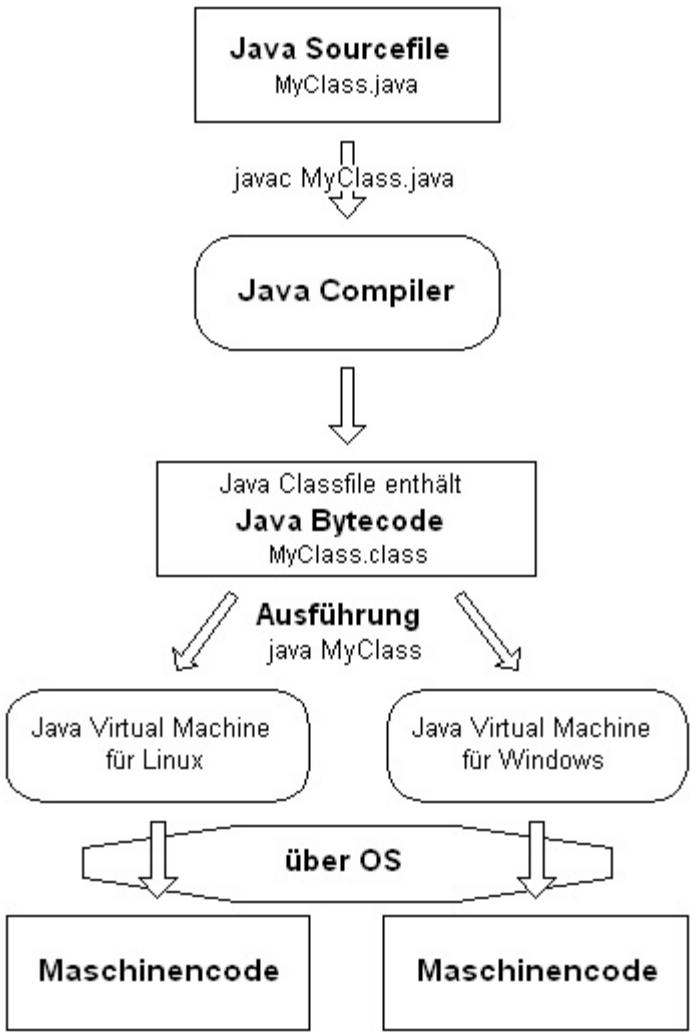


Abbildung 1: Compilation und Plattformunabhänigkeit

Java ist multithreadfähig und dynamisch. In einem Programm können viele Dinge gleichzeitig passieren. Diese Eigenschaft nennt man multithreading. Java ist multithreadfähig und unterstützt den Entwickler mit verschiedenen Funktionen beim Entwurf solcher parallel arbeitender Programme.

Ausführung und Binden. Attribute und Methoden, die nicht überschrieben werden können, werden in Java statisch gebunden. Das gilt für static, private und final Methoden. Alle anderen Methoden werden in der Regel dynamisch gebunden. Jede Klasse mit der Methode:

```
public static void main(String[] args)
```

resultiert in einem ausführbaren Classfile. Beim Laden des Classfiles, wird der Bytecode jeder Methode als Stream in das Java Runtime Environment geladen und von dort weiterverarbeitet. Die JVM führt die Java Programme aus (Abbildung 1).

Opcode und Mnemonic. Ein Befehl der virtuellen Java Maschine besteht aus einem 1 Byte (ohne Vorzeichen) großen Operationscode (Opcode im Java Jargon), der die entsprechende Operation angibt. Diesem 1 Byte Opcode folgen null oder mehrere Operanden. Dabei wird mit dem Opcode die Anzahl und der Typ der Operanden gleich festgelegt. Um die Maschinenbefehle in einer leserlichen Form darstellen zu können, ist jedem Opcode ein Mnemonic (Assembler Sprache der virtuellen Java Maschine) zugeordnet. In jedem Mnemonik ist der Typ des Befehls kodiert. D.h. an Hand des Befehls können die Typen der evtl. notwendigen Argumente abgelesen werden. Der iload Befehl z.B. lädt den Inhalt einer lokalen Variablen, die vom Typ int sein muss, auf den Operandenstapel. Java verfolgt die Strategie, den Code möglichst knapp und Platz sparend zu halten. So ist auch der verhältnismäßig große Befehlssatz der JVM zu erklären. Diese verwendet keine Befehle die verschiedene Operanden verarbeiten können sondern bietet für verschieden Operanden verschiedene Befehle an. Hier kommen die Präfixe der Datentypen zum Einsatz, indem sie im Mnemonic dem Befehl vorangestellt sind (siehe Tabelle 1).

Typ	Buchstabe
byte	b
short	s
int	i
long	l
char	c
float	f
double	d
reference	a

Tabelle 1: Präfixe für Mnemonics

4.2 Struktur des Classfiles

Ein Classfile ist das Resultat der Compilation einer Klasse oder eines Interfaces. Es enthält den plattformunabhängigen Code in Form eines strukturierten (8 Bit) Bytestroms. Worte die größer als 8 Bit sind (Multibyte Daten), werden als 2 oder 4 konsekutive Bytes gelesen. Dies geschieht nach der big endian Regel (high order bytes first). Das Classfile hat eine eigene Menge von Datentypen: u1, u2 und u4 als vorzeichenlose Ein-, Zwei- bzw. Vier-Byte Einheiten. Tabelle 2 gibt die Struktur des Classfiles an. Im weiteren wird auf die Bedeutung der Items eingegangen.

Typ	Item
u4	magic
u2	minor_version
u2	major_version
u2	constant_pool_count
cp_info	constant_pool [constant_pool_count - 1]
u2	access_flags
u2	this_class
u2	super_class
u2	interfaces_count
u2	interfaces [interfaces_count]
u2	fields_count
field_info	fields [fields_count]
u2	methods_count
method_info	methods [methods_count]
u2	attributes_count
attribut_info	attributes [attributes_count]

Tabelle 2: Struktur des Classfiles

4.2.1 Classfile Items

magic
Hierbei handelt es sich um die "`magic Number`". Ein gültiges Classfile beginnt an dieser Stelle immer mit dem Wert 0xCA FE BA BE. Dabei ist die Existenz dieses Wertes keine hinreichende, aber eine notwendige Bedingung bezüglich der Gültigkeit des Classfiles.

minor_version
Gibt den Nachkommaanteil der Version des Classfile Formats an (wird auch als kleine Versionsnummer bezeichnet).

major_version
Der Vorkommaanteil der Version des Dateiformates (große Versionsnummer). Die major_version und die minor_version zusammengesetzt ergibt z.B. 46,0 (0x00 00 00 2E).

constant_pool_count
Der constant_pool_count enthält die Anzahl der Einträge des constant_pool + 1. Das heißt, die Zahl des constant_pool_count ist um eins größer als es Einträge im constant_pool gibt.

constant_pool
Der constant_pool bildet eine Tabelle für verschiedene Strukturen variabler Länge. Solche Strukturen sind z.B. alle verschiedenen Typen von Konstanten einer Klasse sowie die symbolischen Referenzen auf alle verwendeten Klassen, Methoden und Attribute. Der restliche Bytecode des Classfiles, enthält nun Verweise auf den richtigen Index im constant_pool. Auf diese Art ist ein geringer Platzverbrauch durch mehrmalige Verweise auf einen Index möglich. Der Eintrag mit dem Index 0 ist nicht vorhanden (bzw. ist reserviert für den JVM intern Gebrauch). Der constant_pool beginnt also immer entgegen allen Java Konventionen mit dem Index 1. Jeder Eintrag im constant_pool beginnt mit einem Byte (u1), das den Typ des Eintrags angibt (siehe "`tag`" in der Spalte "`Name`" der Tabelle 3). Tabelle 3 gibt die 11 verschiedenen Typen von Konstanten an. In der Spalte "`Wert`" ist die Zahl angegeben, die den Typ der Konstanten klassifiziert (dieser Wert steht also im "`tag`" Feld und identifiziert so den Typ der Konstanten). Die letzen beiden Spalten geben den kompletten Aufbau des Eintrags an, so besteht ein CONSTANT_Integer beispielsweise aus den Feldern "`tag`" und "`bytes`" mit den Typen u1 und u4, wobei im "`tag`" Feld der Wert 3 steht. In dem "`bytes`" Feld würde beispielsweise der Wert 0x00 00 80 E8 stehen, wenn innerhalb des Java Programms einer int Variablen der Wert 33000 zugewiesen wurde (33000 in hexadezimal ergibt 0x00 00 80 E8). Die Eintrage des constant_pool sind wie Tabelle 2 zeigt von dem Typ cp_info. Nachfolgend sollt die Struktur von cp_info angegeben werden.

```
cp_info
{
     u1 tag;
     u1[] info;
}
```

Typ	Wert	Beschreibung	Typ	Name
CONSTANT_Utf8	1	Ein UTF8 codierter Unicode String	u1 u2 u1	tag length bytes[]
CONSTANT_Integer	3	Ein int Wert	u1 u4	tag bytes
CONSTANT_Float	4	Ein float Wert	u1 u4	tag bytes
CONSTANT_Long	5	Ein long Wert	u1 u4 u4	tag high_bytes low_bytes
CONSTANT_Double	6	Ein double Wert	u1 u4 u4	tag high_bytes low_bytes
CONSTANT_Class	7	Eine Referenz auf einen CONSTANT_Utf8 Eintrag, der einen voll qualifizierten Klassen- oder Schnittstellennamen enthält	u1 u2	tag name_index
CONSTANT_String	8	Eine Referenz auf einen CONSTANT_Utf8 Eintrag, der eine Stringkonstante enthält	u1 u2	tag string_index
CONSTANT_Fieldref	9	Eine Referenz auf einen CONSTANT_Class Eintrag, sowie auf einen CONSTANT_NameAndType Eintrag, der Name und Typ eines Attributs der referenzierten Klasse enthält	u1 u2 u2	tag class_index name_and_type_ index
CONSTANT_ Methodref	10	Eine Referenz auf einen CONSTANT_Class Eintrag, sowie auf einen CONSTANT_NameAndType Eintrag, der Name und Typ (Rückgabewert plus Parameter) einer Methode der referenzierten Klasse enthält	u1 u2 u2	tag class_index name_and_type_ index
CONSTANT_Interface Methodref	11	Eine Referenz auf einen CONSTANT_Class Eintrag, sowie auf einen CONSTANT_NameAndType Eintrag, der Name und Typ (Rückgabewert plus Parameter) einer Methode der referenzierten Schnittstelle enthält	u1 u2 u2	tag class_index name_and_type_ index
CONSTANT_Name AndType	12	Eine Referenzen auf zwei CONSTANT_Utf8 Einträge, von denen der erste einen Namen und der zweite einen Typ der jeweiligen Methode bzw. des jeweiligen Attributes enthält	u1 u2 u2	tag name_index descriptor_index

Tabelle 3: Mögliche Einträge im constant_pool

access_flags
Die zwei Bytes des access_flags bildet einen Bitvektor für die Zugriffsrechte, Eigenschaften und Merkmale einer Klasse (siehe Tabelle 4). Betrachtet man die hexadezimale Zahl binär, lassen sich mit Hilfe der Tabelle 5 die Eigenschaften der Klasse ermitteln. Hierbei geht es darum welche Bits gesetzt sind, wenn access_flags den Wert 0x00 21 hat, sind Bit 0 und 5 gesetzt, was laut Tabelle 5 bedeutet, dass die Klasse public ist und der neuen Semantik von invokespecial folgt.

Klassenmerkmale	Bitmaske	Beschreibung
ACC_PUBLIC	0x00 01	Klasse ist public deklariert und kann deshalb außerhalb des Packages verwendet werden
ACC_FINAL	0x00 10	Klasse ist final deklariert und darf somit nicht abgeleitet werden
ACC_SUPER	0x00 20	Methoden der Superklasse werden besonders behandelt. Kompatibilitätsmodus für Classfiles von alten Sun Compilern. Es gibt zwei verschiedenen Semantiken von invokespecial, neue Java Compiler setzen dieses Bit alte VMs ignorieren es.
ACC_INTERFACE	0x02 00	Classfile repräsentiert Interface ACC_ABSTRACT muss gesetzt sein, und außer ACC_PUBLIC darf nichts gesetzt sein
ACC_ABSTRACT	0x04 00	Klasse ist abstract deklariert und kann somit nicht instanziert werden

Tabelle 4: Zugriffsrechte, Eigenschaften und Merkmale von Klassen

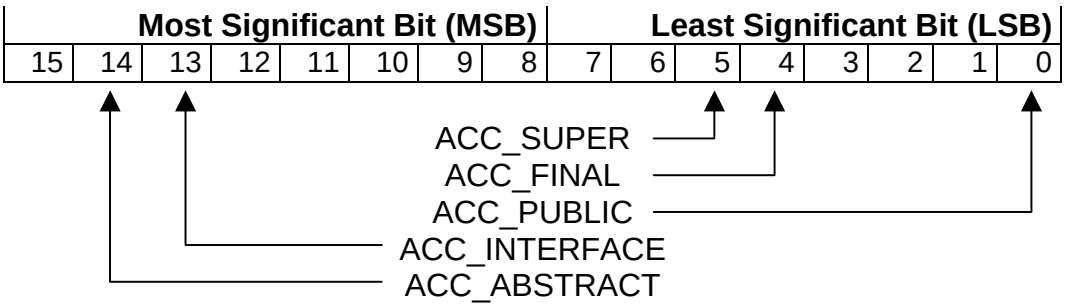


Tabelle 5: Binäre Darstellung des access_flags

this_class
Dieser Eintrag vom Typ u2 ist ein Verweis in den constant_pool. Dort muss an der angegebenen Indexstelle ein Eintrag des Typs CONSTANT_Class stehen, der seinerseits zu einem CONSTANT_Utf8 Eintrag zeigt. Dieser CONSTANT_Utf8 Eintrag enthält dann den Namen dieser Klasse.

super_class
Das Feld super_class verweist ebenso wie this_class auf einen CONSTANT_Class Eintrag und dieser wieder zu einem CONSTANT_Utf8. Der CONSTANT_Utf8 Eintrag enthält nun den Namen der Super Klasse.

interfaces_count
Die zwei Bytes des interfaces_count geben die Anzahl der Einträge in der interfaces Tabelle an. Dies entspricht der Anzahl der implementierten Interfaces.

interfaces
Dieses Feld bildet eine Tabelle von Indizes, die in die constant_pool Tabelle verweisen. Jeder Eintrag, auf den verwiesen wird, enthält Informationen zu diesem Interface und muss von Typ CONSTANT_Class sein. Der CONSTANT_Class Eintrag zeigt wie in den vorigen Fällen (this_class und super_class) seinerseits auf ein CONSTANT_Utf8 Eintrag mit dem Oberinterface (den Klassennamen der implementiert wird).

fields_count
Dieses Feld enthält die Anzahl der Einträge der Tabelle fields. Der Wert entspricht der Anzahl der Attribute der Klasse oder des Interfaces.

fields
Die fields Tabelle enthält Informationen über die deklarierten Attribute der Klasse bzw. des Interfaces, ohne die evtl. geerbten Attribute. Mögliche Informationen sind z.B. Zugriffsrechte, Name und Typ des Attributs. Die Tabelle 6 zeigt den Aufbau der fields Tabelle. Eine genauere Beschreibung der letzten beiden Felder (attributes_count und attributes) wird in Kapitel 4.2.2 erfolgen. An dieser Stelle sei nur erwähnt, dass wenn beispielsweise ein Attribut als final deklariert wurde es zum einem sich im access_flags niederschlägt aber auch zum anderem in der attributes Tabelle (vgl. 4.2.2).

Typ	**Item**		**Beschreibung**
u2	access_flags		Beschreibt die Zugriffsrechte auf ein Attribut
	Eigenschaft	**Maske**	**Beschreibung**
	ACC_PUBLIC	0x00 01	sichtbar für alle
	ACC_PRIVATE	0x00 02	nur für Klasse sichtbar
	ACC_PROTECTED	0x00 04	auch in abgeleiteten Klasse sichtbar
	ACC_STATIC	0x00 08	statisch
	ACC_FINAL	0x00 10	darf nicht überschrieben werden
	ACC_VOLATILE	0x00 40	darf nicht im Cache abgelegt werden
	ACC_TRANSIENT	0x00 80	darf nicht für persistente Objekte verwendet werden
u2	name_index		Index auf constant_pool Eintrag (Utf8) für den Attributnamen
u2	descriptor_index		Index auf constant_pool Eintrag (Utf8) für den Typ des Attributs. Eine ausführliche Beschreibung der Utf8 Zeichenkette erfolgt in Kapitel 4.2.3
u2	attributes_count		Anzahl der zusätzlichen Attribute
attribute_info	attributes [attributes_count]		Attribut Tabelle mit variabler Länge

Tabelle 6: Die fields Tabelle

methods_count
Dieser Eintrag enthält die Anzahl der Einträge der Tabelle methods. Der Wert entspricht der Anzahl der Methoden der Klasse bzw. des Interfaces.

methods
Die methods Tabelle enthält Informationen über die deklarierten Methoden und Konstruktoren der Klasse oder eines Interfaces ohne evtl. geerbte. Mögliche Informationen sind z.B. Zugriffsrechte, Name sowie Typ der Parameter und des Rückgabewerts. Tabelle 7 zeigt den Aufbau der methods Tabelle die der fields Tabelle ziemlich ähnlich sieht. Wie auch schon bei der fields Tabelle erwähnt wird auf die letzten beiden Felder (attributes_count und attributes) in Kapitel 4.2.2 eingegangen. In jeder nicht abstrakten Methode kommt hier jedoch das Code Attribute vor, dass den Bytecode enthält, mehr zum Code Attribute im Kapitel 4.2.2

Typ	**Item**		**Beschreibung**
u2	access_flags		Beschreibt die Zugriffsrechte auf ein Attribut
	Eigenschaft	**Maske**	**Beschreibung**
	ACC_PUBLIC	0x00 01	sichtbar für alle
	ACC_PRIVATE	0x00 02	nur für Klasse sichtbar
	ACC_PROTECTED	0x00 04	auch in abgeleiteten Klasse sichtbar
	ACC_STATIC	0x00 08	statisch
	ACC_FINAL	0x00 10	darf nicht überschrieben werden
	ACC_SYNCHRONIZED	0x00 20	Benutzung muss geschützt werden
	ACC_VOLATILE	0x00 40	darf nicht im Cache abgelegt werden
	ACC_TRANSIENT	0x00 80	darf nicht für persistente Objekte verwendet werden
	ACC_NATIVE	0x01 00	in einer anderen Sprache implementiert
	ACC_ABSTRACT	0x04 00	es gibt keine Implementation
	ACC_STRICT	0x08 00	“floating point“ Modus ist FP strict
u2	name_index		Index auf constant_pool Eintrag (Utf8) für den Methodennamen
u2	descriptor_index		Index auf constant_pool Eintrag (Utf8), der den Methodendescriptor enthält, genauere Beschreibung siehe Kapitel 4.2.3 (Rückgabetyp und Parameter der Methode)
u2	attributes_count		Anzahl der zusätzlichen Attribute
attribute_info	attributes [attributes_count]		Attribut Tabelle mit variabler Länge

Tabelle 7: Die methods Tabelle

attributes_count
Das attributes_count Feld enthält die Anzahl der Einträge in der attributes Tabelle.

attributes
Die attributes Tabelle enthält Attribute der Klasse bzw. des Interfaces z.B. das SourceFile Attribute (siehe Kapitel 4.2.2). Eine Beschreibung erfolgt im nächsten Kapitel.

Ein Überblick über die wesentlichen Teile des Classfile Formats verschafft die Tabelle im Anhang A.

4.2.2 Attribute

Attribute können sowohl bei Klassen als auch bei Feldern und Methoden sowie als Unterbestandteil des Attributs "`Code Attribute`" (siehe Kapitel 4.2.2) vorkommen. Es gibt drei verschiedene Arten von Attributen. Die eine Art muss von allen VMs erkannt werden, eine weitere soll von allen VMs erkannt werden und als drittes besteht die Möglichkeit für Programmierer von Java Compilern, beliebige eigene Attribute einzuführen. Sie müssen sich dafür an die Namensgebung von Sun's "`Reverse Domain Name`" Notation halten. Dabei darf die Ausführung des Programms nicht von diesen neuen Attributen abhängig sein, dass heißt, auf einer VM, die das Attribut nicht kennt, muss das Programm trotzdem genau das tun, was von ihm erwartet wird (trifft eine VM auf ein ihr unbekanntes Attribut, muss sie es ignorieren). Ein Beispiel für ein neues Attribut wäre z.B. eines, welches die Version des Compilers enthält, mit dem es compiliert wurde.
Jede Art von Attribut ist durch ihren Namen eindeutig definiert und kann einen beliebigen Inhalt haben. Damit dies möglich ist, haben alle Attribute zu Beginn einen gleichen Aufbau, dieser ist in Tabelle 8 beschrieben.

Typ	**Item**	**Beschreibung**
u2	attribute_name_index	Verweis in den constant_pool auf einen Utf8 Eintrag, der den Namen des Attributes enthält
u4	attribute_length	Anzahl der noch folgenden Bytes des Attributs

Tabelle 8: Grundsätzlicher Aufbau eines Attributs

Der erste short (u2 - attribute_name_index) eines Attributs ist immer ein Index in den constant_pool, der auf einen UTF8 Eintag mit dem Attributnamen verweist – auf diese Weise ist der Typ definiert (z.B. "`Code`", "`ConstantValue`" oder "`SourceFile`").

Attribute, die von allen VMs erkannt werden müssen

Code Attribute

Das “Code“ Attribut enthält den eigentlichen Java Bytecode und muss in jeder nichtabstrakten Methode vorhanden sein. Sein Aufbau ist Tabelle 9 zu entnehmen.

Typ	Item	Beschreibung
u2	attribute_name_index	Index auf Utf8 Eintrag im constant_pool, enthält “Code“
u4	attribute_length	Anzahl der noch folgenden Bytes des Attributs
u2	max_stack	Maximale Größe des (Operanden) Stacks
u2	max_locals	Maximale Anzahl an lokalen Variablen
u4	code_length	Länge des Bytecodes in Byte
u1	code[code_length]	Bytecode der Methode
u2	exception_table_length	Anzahl der Einträge in der exception_table Tabelle
exception_table	exception_table[exception_table_length]	Exception Tabelle, der Aufbau wird in Tabelle 10 beschrieben
u2	attributes_count	Anzahl weiterer Attribute in der attributes Tabelle
attribute_info	attributes [attributes_count]	Tabelle mit weiteren Attributen möglich (LineNumberTable Kap. 4.2.2 und LocalVariableTable Kap. 4.2.2)

Tabelle 9: Das Code Attribute

Die exception_table beinhaltet sämtliche Informationen für Exception Behandlungen einer Methode. Der Aufbau ist in Tabelle 10 dargestellt.

Typ	Item	Beschreibung
u2	start_pc	Anfang des zu überwachenden Bereichs im Bytecode (Offset relativ zum Methodenanfang); Startpunkt der Gültigkeit des Exception-Handlers
u2	end_pc	Ende des zu überwachenden Bereichs im Bytecode (relativ zum Methodenanfang); Endpunkt der Gültigkeit des Exception-Handlers
u2	handler_pc	Adresse des Startpunkts des Exception-Handlers im Bytecode (catch-Statement)
u2	catch_type	Referenz auf einen CONSTANT_Class Eintrag im constant_pool, der seinerseits auf einen CONSTANT_Utf8 Eintrag zeigt. Der UTF8 Eintrag enthält den Klassennamen, der die zu fangende Exception darstellt

Tabelle 10: Die exception_table

ConstantValue Attribute

Das "`ConstantValue`" Attribut ist nur für statische Felder gültig. Wird ein Feld mit einem konstanten Wert definiert, so schreibt der Compiler den Wert dieses Feldes in dieses Attribut. Aufbau siehe Tabelle 11.

Typ	Item	Beschreibung
u2	attribute_name_index	Index auf Utf8 Eintrag im constant_pool, enthält "ConstantValue"
u4	attribute_length	Anzahl der noch folgenden Bytes des Attributs
u2	constantvalue_index	Index zu einem Long-, Integer-, Double-, Float- ... Eintrag in dem constant_pool mit dem Wert

Tabelle 11: Das ConstantValue Attribute

Exceptions Attribute

Das Attribut "`Exceptions`" enthält eine Liste aller Exceptions, die eine Methode nach außen hin werfen kann in Form von Referenzen auf den constant_pool. Aufbau siehe Tabelle 12.

Typ	Item	Beschreibung
u2	attribute_name_index	Index auf Utf8 Eintrag im constant_pool, enthält "Exceptions"
u4	attribute_length	Anzahl der noch folgenden Bytes des Attributs
u2	number_of_exceptions	Anzahl der Einträge in der exception_index_table. Dies entspricht der Anzahl der Exception nach außen
u2	exception_index_table[number_of_exceptions]	Class Eintrag im constant_pool mit Verweis auf Klasse der Exception

Tabelle 12: Das Exception Attribute

InnerClasses Attribute

Das Attribut "`InnerClasses`" enthält Namen und Flags aller Klassen, die Member Klassen der aktuellen Klasse sind. Member Klassen sind Klassen, die innerhalb einer anderen Klasse definiert wurden (Innerclass).

Synthetic Attribute

Das Vorhandensein des Attributs "`Synthetic`", welches keine eigenen Eigenschaften hat (also die Funktion eines Flags erfüllt), zeigt, dass das Attribut, die Methode oder die Klasse, bei der es steht, nicht vom Programmierer sondern vom Compiler generiert wurde.

Sun Standard Attribute, die nicht zwangsläufig erkannt werden müssen

Deprecated Attribute

Das Attribut "`Deprecated`" hat keine Eigenschaften, es erfüllt also ebenso wie das Synthetic Attibute die Funktion eines Flags. Ein Compiler kann dieses Attribut, welches sowohl in Klassen als auch in Feldern und Methoden vorkommen kann, verwenden, um darauf hinzuweisen, dass der Programmierer ein Element der API verwendet, welches offiziell nicht mehr verwendet werden sollte.

LineNumberTable Attribute

Dieses Attribut, welches als Unterattribut von "`Code`" vorkommen kann, kann die VM für Fehlermeldungen, wie zum Beispiel, aus welcher Quellcodezeile welche Stücke des Bytecodes ursprünglich stammten, verwenden. Diese Information kann vom Debugger und Exception Handler weiter benutzt werden. Den Aufbau zeigt Tabelle 13.

Typ	Item	Beschreibung
u2	attribute_name_index	Index auf Utf8 Eintrag im constant_pool, enthält "LineNumberTable"
u4	attribute_length	Anzahl der noch folgenden Bytes des Attributs
u2	line_number_table_length	Anzahl der Einträge in der line_number_table Tabelle
line_number_table	line_number_table[line_number_table_length]	Aufbau siehe Tabelle 14

Tabelle 13: Das LineNumberTable Attribute

Die line_number_table Tabelle ordnet einem Bytecode Teil eine Zeile aus der Quelldatei zu. Der Aufbau ist in Tabelle 14 dargestellt.

Typ	Item	Beschreibung
u2	start_pc	Offset im Bytecode ab dem die Zeile beginnt
u2	line_number	Zeilennummer in der Quelldatei

Tabelle 14: Die line_number_table

LocalVariableTable Attribute

Dieses Attribut assoziiert lokale Variablen (welche im Classfile keine Namen mehr haben, sondern nur über einen Index identifiziert werden) mit ihren ursprünglichen Namen (als UTF8-Einträge im Konstantenpool). Es ist ebenfalls dazu gedacht, aussagekräftige Fehlermeldungen zu erzeugen. Den Aufbau der LocalVariableTable Tabelle zeigt Tabelle 15.

Typ	Item	Beschreibung
u2	attribute_name_index	Index auf Utf8 Eintrag im constant_pool, enthält “LocalVariableTable“
u4	attribute_length	Anzahl der noch folgenden Bytes des Attributs
u2	local_variable_table_length	Anzahl der Einträge in der local_variable_table Tabelle
local_variable_table	local_variable_table[local_variable_table_length]	Aufbau verdeutlicht Tabelle 16

Tabelle 15: Das LocalVariableTable Attribute

Die local_variable_table wird vom Debugger benutzt, um den Wert und den Namen einer lokalen Variable zu ermitteln. Ihren Aufbau zeigt Tabelle 16.

Typ	Item	Beschreibung
u2	start_pc	Offset ab dem die Variable einen Wert hat
u2	length	Länge dieses Bereichs
u2	name_index	Utf8 Eintrag: Name der Variable
u2	descriptor_index	Utf8 Eintrag: Typ der Variable (Descriptoren siehe Kapitel 4.2.3)
u2	index	Index der Variable im Frame der Methode

Tabelle 16: Die local_variable_table

SourceFile Attribute

Das Attribut “`SourceFile`“ identifiziert einen UFT8-Eintrag im constant_pool, der den original Dateinamen, aus dem die Klasse generiert wurde, enthält. Aufbau siehe Tabelle 17.

Typ	Item	Beschreibung
u2	attribute_name_index	Utf8 Eintrag im constant_pool enthält “SourceFile“
u4	attribute_length	Anzahl der noch folgenden Bytes, des Attributs
u2	sourcefile_index	Verweist auf einen Utf8 Eintrag, der den Namen der Java-Quellcodedatei enthält

Tabelle 17: Das SourceFile Attribute

4.2.3 Descriptoren

Descriptoren werden für die interne Darstellung von Datentypen benutzt. Dies geschieht unter anderem in den Items fields und methods (vgl. Kapitel 4.2.1), also bei Attributen und Methoden. Handelt es sich um ein Attribut, wird der Typ angegeben. Im Fall einer Methode werden die Typen von Rückgabewert und Parametern spezifiziert (siehe Tabelle 6 und 7). Aus Gründen der Ausführlichkeit wird nachfolgend zuerst das Typsystem von Java angegeben. Evtl. im Soucecode auftauchende parametrisierte Typen sind an dieser Stelle (wenn der Compiler die Descriptoren erzeugt) schon umgewandelt worden (hierzu siehe Kapitel 5.3). Das Java Typsystem:

Type	→	PrimitiveType | ReferenceType
PrimitiveType	→	NumericType | boolean
NumericType	→	IntegralType | FloatingPointType
IntegralType	→	byte | short | int | long | char
FloatingPointType	→	float | double
ReferenceType	→	ClassOrInterfaceType | ArrayType
ClassOrInterfaceType	→	ClassType | InterfaceType
ClassType	→	TypeName
InterfaceType	→	TypeName
ArrayType	→	Type[]

An dieser Stelle noch eine Bemerkung zum Classfile, innerhalb der CONSTANT_Utf8 Einträge wird "/" statt "." als Trennzeichen für den Paketpfad benutzt wird. Außerdem lösen Java Compiler nicht qualifizierte Pfade vollständig auf. Hierzu ein Beispiel:

Sourcefile:	`package mypackage;` `class Address extends Object{...}`
Umwandlung:	`class mypackage/Address extends java/lang/Object`

Mit diesem Hintergrund wird nun die Grammatik des Deskriptors für Attribute angegeben. Tabelle 18 verdeutlicht die letzten drei Zeilen der Grammatik. Daran schließen sich Beispiele an.

Die Grammatik für Deskriptoren, die Attribute betrefffen:

FieldDescriptor	→	FieldType
ComponentType	→	FieldType
FieldType	→	BaseType | ObjectType | ArrayType
BaseType	→	B | C | D | F | I | J | S | Z
ObjectType	→	**L**<classname>**;**
ArrayType	→	**[**ComponentType

Interne Darstellung	**Typ**	**Interpretation**
B	byte	8 Bit Wert mit Vorzeichen
C	char	16 Bit Zeichen unicode-kodiert
D	double	Fließkommazahl, 64 Bit
F	float	Fließkommazahl, 32 Bit
I	int	ganze Zahl, 32 Bit
J	long	ganze Zahl, 64 Bit
S	short	ganze Zahl, 16 Bit
Z	boolean	true oder false
L<classname>**;**	reference	Instanze der Klasse <classname>
[	reference	eine Dimension eines Arrays

Tabelle 18: Der BaseType, ObjectType und ArrayType

Bespiele:

Sourcecode	**Umwandlung**	**Descriptor**
boolean b;		Z
Object o;	java/lang/Object	Ljava/lang/Object;
Address[][] a;	mypackage/Address	[[Lmypackage/Address;
int i;		I
char[] c;		[C
MyClass x		LMyClass;

Bei Methoden kommt eine beliebige Anzahl von Parametern (*) und der Rückgabewert hinzu. Die Grammatik sieht dann wie folgt aus:

MethodDescriptor	→	(ParameterDescriptor*)ReturnDescriptor
ParameterDescriptor	→	FieldType
ReturnDescriptor	→	FieldType | V
FieldType	→	BaseType | ObjectType | ArrayType
BaseType	→	B | C | D | F | I | J | S | Z
ObjectType	→	**L**<classname>**;**
ArrayType	→	**[**ComponentType

Wobei der ReturnDescriptor entweder ein FieldType oder V → void sein kann.

Beispiele:

Sourcecode	**Umwandlung**	**Descriptor**
void main(String[] args)	java/lang/String	([Ljava/lang/String;)V
short[] method(int a, Object b)	java/lang/Object	(ILjava/lang/Object;)[S
void dummy(int a, long b, float c)		(IJF)V
void codegen(ClassFile a, CodeAttribute b)		(LClassFile;LCodeAttribute;)V
Vector getV()	java/util/Vector	()Ljava/util/Vector;

4.3 Entwicklungsumfeld

Für die Entwicklung des Compiler wurde hauptsächlich mit Eclipse [ECL03] gearbeitet, aus diesem Grund wird an dieser Stelle kurz auf die umfassenden Funktionen von Eclipse eingegangen, die in Bezug auf das Projekt auch genutzt wurden.

- **Leistungsstarker Editor**
  - Fehleranzeige
    Der gerade editierte Quelltext wird in Eclipse während des Speichervorgangs übersetzt und auftretende Fehler werden im Quelltext sofort markiert. Diese können auf diese Weise schnell lokalisiert und verbessert werden. Sämtliche Fehler werden zusätzlich in einer Übersichtstabelle dargestellt und es kann schnell an die jeweilige Stelle gesprungen werden. Außerdem werden Fehler auch währen des Schreibens direkt angezeigt (ohne Speichervorgang).
  - Try-catch Blöcke
    Fehlende try-catch Blöcke lassen sich recht komfortabel über die Option "Surround with try / catch block" einfügen. Nach dem Markieren des entsprechenden Codeblockes werden alle Exceptions, die in diesem Block entstehen können, mit dem jeweiligen Catch-Statements gefangen.
  - Code Templates
    Jede Entwicklungsumgebung stellt heutzutage eine Tag-Completion Möglichkeit zur Verfügung. Tippt ein Anwender den Anfang eines Statements, werden sofort mögliche Vervollständigungen angeboten. In Eclipse können so normale Java Statements sowie JavaDoc Kommentare eingefügt werden.
    Natürlich bietet Eclipse in diesem Zusammenhang auch die Möglichkeit Methoden zum Überschreiben von Basisklassenmethoden zu generieren und Getter- Setter- Zugriffsmethoden für Attribute zu erzeugen.
- **komfortable Suchfunktionalität**
  Es kann in ’normalen’ Textdateien, Hilfsdateien oder Java Quelltextdateien gesucht werden (auch in allen Projektdateien auf einmal). Innerhalb der Java Dateien hat man unter anderem die Möglichkeit gezielt nur nach Deklarationen einzelner Klassen oder Attribute und ähnliches zu suchen. Die Ergebnisse werden in einer Liste dargestellt, und können somit leicht angesprungen werden.
- **umfangreiches Vergleichswerkzeug**
  Eclipse bietet mit einem umfangreichen Vergleichswerkzeug, mit dem es möglich ist zwei Dateien oder zwei Projekte miteinander zu vergleichen. Die Unterschieden werden in einer sehr übersichtlichen Form dargestellt.
- **CVS Unterstützung**
  Mit Eclipse bekommt man einen leistungsstarken CVS Client, mit dem man sehr leicht Projekte auschecken, bearbeiten und einchecken kann. Außerdem werden dem Benutzer die Unterschiede seiner lokalen Kopie mit der Version auf dem CVS Server über das zuvor erwähnte Vergleichswerkzeug angezeigt.

- **Debugger**
  Die Debug Perspective in Eclipse stellt umfangreiche Debugging-Funktionalität zur Verfügung und ermöglicht eine gute Übersicht über den aktuellen Zustand der virtuellen Maschine. Durch die Evaluierung von Einzelstatements oder eine Überprüfung von Attributen und lokalen Variablen lassen sich viele weitere Informationen über die VM erhalten.
- **Weitere Vorteile**
  - Das Eclipse Projekt ist ein Open Source Projekt, und somit ist die IDE auch im kommerziellen Bereich kostenlos.
  - Außerdem werden dem Benutzer eine Vielzahl an Plug Ins angeboten. In diesem Projekt wurde ein UML Plug In von Omondo [OMO04] eingesetzt (siehe Kapitel 4.4.4).

Außer Eclipse wurde zur Analyse der generierten Classfiles verschiedene Disassembler verwendet (jvmDisassembler [PLÜ03], ClassCracker [MAY03] und jad [KOU03]). Das hatte den Hintergrund, dass die verschiedenen Disassembler verschieden mit Fehlern in den Classfiles umgehen. So konnte festgestellt werden, dass ein Disassembler das disassemblieren wegen eines Fehlers im Classfile abbrach, ein anderer aber noch fertig disassembliert, oder eine konkrete Fehlermeldung ausgab. Eine grundsätzliche Aussage, welchem Disassembler Vorzug zu geben ist, kann aber nicht getroffen werden, da das Verhalten von der Fehlerart abhängig war. Außerdem wurde ein Hexeditor eingesetzt, um die Classfiles zu analysieren und gegebenenfalls zu manipulieren. Dabei war auch die im Anhang A befindliche Tabelle über das Classfile Format hilfreich. Als Versionsverwaltungssoftware kam wie schon angesprochen CVS [HEL04] zum Einsatz. Außerdem wurde ein Linux Derivat benutzt um jay [KÜH04b] und make [FRE04] nutzen zu können. JLex [KÜH04a] wurde als Scanner Generator eingesetzt.

4.4 MyCompiler

4.4.1 Compilation

Zum compilieren des Compiler liegt ein `Makefile` vor, welches unter Linux mit dem Befehl `make` ausgeführt wird. Alle nötigen Programme befinden sich ebenfalls im Sourcecode Verzeichnis (JLex [KÜH04a] und jay [KÜH04b]). Herr Ott hat außerdem die *.exe, *.bat, *.dll sowie die *.cygwin Dateien ins das Sourcecode Verzeichnis des Compiler eingecheckt. Genauere Informationen zu diesen Dateien sind seiner Studienarbeit [OTT04] zu entnehmen.

4.4.2 Aufruf des MyCompiler

Der Compiler wird mit folgendem Befehl aufgerufen:

```
java MyCompiler <File> [Debug Level]
```

Hierbei ist es möglich einen Debug Level anzugeben, der die Menge der Debug Ausgaben begrenzt. Wenn kein Debug Level angegeben wurde, sollten auch keine Debug Ausgaben angezeigt werden. Welche Debug Ausgaben wie eingeschaltet werden erfährt man mit dem Aufruf:

```
java MyCompiler
```

An dieser Stelle noch der Hinweis, dass zum Teil die Debug Ausgaben auch über Attribute in den Klassen gesteuert werden kann. Dies hat zum einem den Grund, dass diese zum Teil nicht auf lange Sicht im Sourcecode bleiben sollen, zum andern werden evtl. heikle Operationen durchgeführt, auf die der Nutzer aufmerksam gemacht werden soll. Solche Attribute sind in den Klassen `ClassFile (hamaDebug, system_out und system_out2), IfStmt und JVMCode` enthalten.

4.4.3 Syntax parametrisierter Klassen im MyCompiler

Die Klassendeklaration mit Parametern

Die Parameterdeklaration erfolgt mit der Deklaration der Klasse. Parameter werden nach dem Klassennamen in spitzen Klammern `<...>` angegeben. Mehrere Parameter werden durch Kommata getrennt. Ein Beispiel hierzu zeigt Listing 2.

```
class MyClass<A, B>
{
     ...
}
```

Listing 2: Klassendeklaration mit Parametern

Soll von einer parametrisierten Klasse geerbt werden, so sind bei der Angabe der Basisklasse die Parameter mit anzugeben. Siehe Listing 3.

```
class Child extends MyClass<A, B>
{
     ...
}
```

Listing 3: Klassendeklaration bei Vererbung

Wenn eine Child Klasse wiederum Parameter enthalten soll, so werden diese wieder nach dem Klassennamen angegeben. Ebenfalls können bei Vererbung die Parameter der Basisklassen festgelegt werden (vgl. Listing 4).

```
class NewChild<C> extends MyClass<A, B>
{
     ...
}
```

Listing 4: Klassendeklaration mit Parametern bei Vererbung

Selbstverständlich können als Parameter auch Klassen angegeben werden, die selbst wieder Parameter besitzen. Das veranschaulicht Listing 5.

```
class ClassA<MyClass<A, B>>
{
     ...
}
```

Listing 5: Klassendeklaration mit einer Klasse als Parametern

Attribut- und Variabelendeklaration

Soll ein parametrisiertes Attribut oder – Variable deklariert werden, so erfolgt dies durch eine Voranstellung des entsprechenden Platzhalters. Siehe hierzu Listing 6.

```
class MyClass<A, B>
{
     A dummy;
     B value;
     ...
}
```

Listing 6: parametrisierte Attributdeklaration

Methodendeklaration

Wenn eine Methode einen parametrisierbar Rückgabewert besitzt, so wird der entsprechende Platzhalter an die Stelle des Rückgabetyps angegeben. Die Angabe von parametrisierbaren Übergabeparameter ist nicht implementiert. Ein Beispiel zeigt Listing 7.

```
A method(A dummy)
{
     ...
}
```

Listing 7: parametrisierbare Methoden

Instanz einer parametrisierten Klasse anlegen
Soll eine Instanz einer parametrisierten Klasse erzeugt werden, so sind die Typen, die übergeben werden sollen, in eckigen Klammern <...> hinter dem Klassennamen anzugeben. Ein Beispiel wird in Listing 8 gegeben.

```
class Name()
{
     void methodX()
     {
          MyClass<String, Integer> varname = new MyClass();
     }
}
```

Listing 8: Instanz einer parametrisierten Klasse anlegen

4.4.4 Klassendiagramme

Im Root Verzeichnis des Compilers sind außer den bereits erwähnten Dateien außerdem *.ucd Dateien enthalten. Diese Dateien enthalten Klassendiagramme, die mit Eclipse bearbeitet werden können. Dazu ist allerdings das bereits erwähnte Plug In [OMO04] nötig. In Verbindung mit den *.ucd Dateien steht auch das .uml Verzeichnis, hier legt das Plug In Verwaltungsinformationen ab, ohne die die Klassendiagramme nicht weiter bearbeitet werden können. Zusätzlich sind Klassendiagramme in Form von pdf's, jpg's und svg's (Scalable Vector Graphics, kann über ein Plug In von Adobe [ADO04] im Browser betrachtet werden) in dem Verzeichnis Rest enthalten.

4.4.5 Inhalt des Rest Verzeichnisses

Das Rest Verzeichnis enthält neben den gerade erwähnten Klassendiagrammen, verschiedene gepackte Versionen des Compilers. Die Version Compiler1.zip ist die Version, die übergeben wurde. Versionen mit höherem Index sind Zwischenstände der Entwicklung. Außerdem sind verschiedene Bücher enthalten:

- CVS, Björn A. Zeeb, 8. Januar 2002
- The Java Virtual Machine Specification, CHAPTER 4, The class File Format
- Adding Generics to the Java Programming Language, Participant Draft Specification, April 27, 2001
- Erweiterung der semantischen Analyse des Java Compilers, Felix Reichenbach, Mai 2003
- The Java Virtual Machine Specification, Second Edition

Zusätzlich ist der erwähnte jvmDisassembler [PLÜ03] in diesem Verzeichnis enthalten.

5 Umsetzung

5.1 Funktionsanalyse des Compilers

Mit Hilfe der Use Cases wurden folgende Ergebnisse bei der Funktionsanalyse des Compilers ermittelt:
1. Es ist nicht möglich, dass in einer Klasse, eine Methode und ein Attribut den gleichen Namen haben.
2. Ebenso ist es nicht möglich, dass es in einer Klassen zwei Methoden mit den gleichen Namen und unterschiedlichen Rückgabewerten gibt (überladen). Zwei oder mehr Konstuktoren sind auch nicht möglich.
3. Modifiers vor Klassen und Methoden werden nicht berücksichtigt (alle werden als public behandelt). Bei Attributen ist es nicht nur so, dass sie nicht berücksichtig werden, sie führen zu einer Exception und die weitere Compilation wird abgebrochen.
4. Im Vererbungsfall werden alle Merkmale der Basisklasse in der abgeleiteten Klasse als public Merkmale derselbigen behandelt.
5. Desweiteren ist es nicht möglich Klassen aus anderen Bibliotheken (z.B. /lib/*oder *.jar) einzubinden.

Folgendes wurde getestet und hat funktioniert:

- Arithmetische Operatoren (+ - * / % ++ --)
- Relationale Operatoren (== != < <= > >=)
- Logische Operatoren (! || & | ^)
- Zuweisungsoperator (=)
- if-Anweisung
- while-Schleife
- boolean
- char
- int
- String
- return
- void

Was nicht funktioniert:

- byte
- short
- long
- float
- double
- Zuweisungsoperatoren (+= -= *= /= %= &= |= ^= <<= >>= >>>=)
- Logische Operatoren (&&)
- instanceof
- Kommentare
- Arrays
- for-Schleife
- Interfaces
- do-Schleife

- Bitweise Operatoren (~ >> >>> <<)
- Fragezeichenoperator
- switch-Anweisung
- break
- continue
- this
- Exceptions (try catch throw finally)
- Integer.MAX_VALUE
- Integer.MIN_VALUE
- Package
- Import
- final
- transient
- volatile
- synchronized
- public
- private
- protected
- abstract
- static

5.2 Eventuelle Bugfix oder Erweiterungen

Dieser Abschnitt beschäftigt sich mit den Ergebnissen des Compiler Tests über die Use Cases. Alle hier angesprochenen Use Cases sind im hama Verzeichnis enthalten und im Anhang abgedruckt. Im dem hama Verzeichnis sind außerdem class Dateien enthalten, die zum einen von verschiedenen Compilern stammen und verglichen wurden, zum anden sollen sie aber auch Zustände des MyCompilers dokumentieren. Bei den class Dateien, die mit einem ein –S im Dateinamen enthalten, haldelt es sich um Dateinen, die von Sun Compiler übersetzt wurden. Wohingegen bei Dateien, die –mx (wobei x hier für eine laufende Nummer steht) im Dateinamen enthalten vom MyCompiler übersetzt wurden.
Die beiden Use Cases Empty und Add (Anhang B) wurden schon zu beginn richtig übersetzt. Die drei Use Cases Ergebnis (Anhang B), Setze und Toggle1 (Anhang C) dagegen nicht. Hier wurde in Absprache mit dem Betreuer beschlossen den Bug, der für die falsche Übersetzung von Toggle1 verantwortlich ist, zu finden und schließlich zu fixen.

5.2.1 Toggle1 - If Bug

Um den Bug, der zu einer falschen Übersetzung des Toggle1 Use Cases führte lokalisieren zu können, wurden systematisch die beiden Classfiles (das eine vom Sun und das andere vom MyCompiler compiliert) verglichen. Nach dem zu Beginn die Classfiles sehr unterschiedlich wirkten, reduzierten sich die vermeintlichen Unterscheide schließlich auf den Codeteil der Methode toggle(). Das lag daran, dass am Anfang zwei Eigenschaften der Classfiles bzw. des Sun Compiler noch nicht klar waren.

1. Die Option –g:none beim compilieren mit dem Sun Compiler lässt den Compiler keine Debugging Informationen ins Classfile schreiben (Aufruf: javac –g:none MyClass.java). Anfänglich wurde ohne diese Option übersetzt, somit wurden Debug Informationen in das Classfile geschrieben, die der MyCompiler nicht hineinschreibt, dadurch unterschieden sich die Classfiles beträchtlich (auch in ihrer Größe).

Nach einer erneuten Compilation mit dieser Option wirkten die Classfiles schon wesentlich 'verwandter'. Jedoch fiel auf, dass die Reihenfolge der Einträge im constant_pool und die Reihenfolge der Methoden unterschiedlich war. Dies ist der zweite Punkt, der anfänglich noch nicht klar war.

2. Die Reihenfolge der Einträge im constant_pool ist irrelevant, gleiches gilt für die Reihenfolge von Methoden und Attributen sofern alle Einträge auf die richtigen constant_pool Einträge verweisen.

Zurück zum erkannten Unterscheid im Codeteil der Methode toggle(). Hier hat das Classfile vom MyCompiler zwischen dem ireturn ($7) und dem iconst_1 ($11) noch ein goto ($8), siehe Abbildung 2. Dieses goto macht keinen Sinn, da es nie erreicht werden kann, denn entweder ist die VM im if-Block und gelangt dann (vor dem goto) zum ireturn und verlässt somit die Methode oder die VM gelangt in den else-Block und springt dann gleich zum iconst_1 (von $1 zu $11).

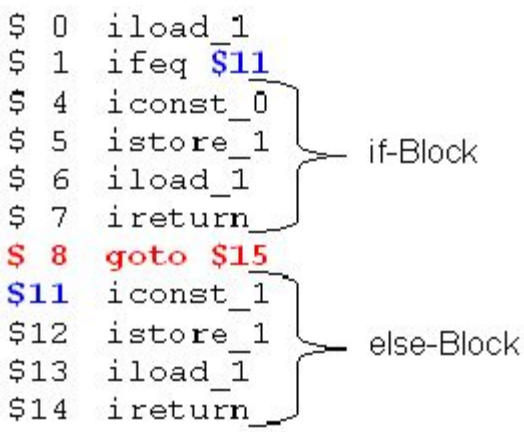


Abbildung 2: Codeteil der Methode toggle() mit falschen goto Befehl

Bemerkung: In diesem Kontext macht das goto keinen Sinn, aber dass nur, weil ein return vor dem goto zwingend erreicht wird. Wäre davor kein return, müsste das goto für das Überspringen des else-Blocks sorgen.
Nachdem diese Erkenntnisse gesammelt wurden, musste die betreffende Soucecode Stelle gefunden und eine zusätzliche Abfrage eingefügt werden. Diese Abfrage sollte anhand voriger returns klären, ob ein goto ins Classfile geschrieben wird oder nicht. Die Klassen die mit der beschriebenen Funktionalität erweitert wurde ist die class IfStmt. Die Änderungen wurden im Sourcecode kommentiert.

5.2.2 Object- und byte-Bug

Beim testen des Compilers mit Klassen, die verschiedene Basistypen oder Objekttypen als Attribute hatten viel auf, dass im constant_pool die Descriptoren nicht richtig lauteten. Dieser Fehler trat immer dann auf, wenn es um den Typ Object oder byte ging. Der MyCompiler erzeugte hier entweder "`LObject;`" oder "`Lbyte;`" anstatt "`Ljava/lang/Object;`" und "B". Eine weiterer Analyse des Problems zeigte, das die Klasse `JVMCode` und insbesondere die Methode `get_codegen_Type` als zentrale Schnittstelle für die Descriptorenbildung dient. Dieser Methode werden alle vorkommenden Typen des Sourcecodes übergeben, ihre Aufgabe ist es dann, diese Typen in die passende Descriptor - Syntax zu übersetzen.

Veranschaulichung:

```
public static String get_codegen_Type(String type, Vector
paralist)
{
     if(type.equals("void")) return "V";
     else if(type.equals("char")) return "C";
     else if(type.equals("double")) return "D";
     else if(type.equals("byte")) return "B";
     ...
     else if(type.equals("Object")) return
          "Ljava/lang/Object;";
     else return "L" + type + ";";
}
```

Die Zeilen byte und Object (rot markierte Einträge) haben in dieser Methode gefehlt, was zu obrigen Auswirkungen führte. Nun werden die korrekten Typen in der Descriptor - Syntax zurückgegeben.

5.3 Implementierung der Codeerzeugung

Die ersten Überlegungen die Codegenerierung für parametrisierte Klassen zu implementieren war der Grundgedanke, dass diese nur geringfügig zur der 'normaler' Klassen abweichen dürfte. Als erster Ansatz sollte ein einfacher Use Case, der keine Typparameter besitzt und den der MyCompiler korrekt übersetzt gefunden werden. Dieser Use Case sollte dann so modifiziert werden, dass alle vorigen ('feste') Typen von Attributen, Rückgabewerten und Parametern parametrisiert werden. Eine Umsetzung dieser Forderungen veranschaulichen Listing 9 linker und rechter Teil (die Use Cases sind im hama Verzeichnis enthalten).

```
01 class StoreSomethingStr
02 {
03   String something;
04
05   String get()
06   {
07        return something;
08   }
09
10   void set(String some)
11   {
12        something = some;
13   }
14 }
```

```
01 class StoreSomethingPar<A>
02 {
03   A something;
04
05   A get()
06   {
07        return something;
08   }
09
10   void set(A some)
11   {
12        something = some;
13   }
14 }
```

Listing 9: Klasse StoreSomethingStr und Klasse StoreSomethingPar

Als nächsten Schritt sollten die Klassen mit den drei verschiedenen Compilern (Sun Compiler, Pizza Compiler [PIZ03] und MyCompiler) compiliert werden (sofern möglich). Ziel sollte es sein, aus den Unterschieden der resultierenden Classfiles abzuleiten, welche Funktionalität im MyCompiler noch implementiert werden muss. In der praktischen Umsetzung wurde dann allerdings mit einer weiteren Klasse gearbeitet, die dem linken Teil von Listing 9 mit dem Typ Object anstatt String entspricht. Dies hatte den Hindergrund genauer zu verstehen, wie sich eine Änderung des Typs auf das Classfile niederschlägt (nichts anderes steht in einer parametrisierten Klasse $\rightarrow$ ein 'anderer' Typ). Die aus einer genauen Analyse gewonnenen Informationen sollten dabei helfen die Bytecodegenerierung für Typparameter zu implementieren.

Bei der Analyse der verschiedenen Classfiles sind folgende Unterschiede aufgefallen:

1. Die Verweise der Deskriptoren von der `StoreSomethingObj` Klasse in den constant_pool, die mit dem MyCompiler compiliert wurde, wurden nicht korrekt generiert. Siehe hierzu Kapitel 5.2.2
2. Beim Pizza Compiler tauchen die Deskriptoren der Typparameter im constant_pool als `Ljava/lang/Objekt;` auf. Beim MyCompiler als `L<Typparametername>;` also im Fall unseres Use Case als `LA;` dies macht keinen Sinn, da das bedeutet, dass der Typ eine Instanz der Klasse A ist (dies ist aber nicht gemeint). Dies ist also ein Punkt wo eingegriffen werden muss um die Übersetzung parametrisierter Klassen zu implementieren
3. Im Konstruktor[2] und in der set Methode der Use Case fehlte ein aload_0. Eine genauere Analyse dieses Problems hat ergeben, dass dieses Problem bei allen Konstruktoren auftritt. An dieser Stelle wird deshalb ein Bug vermutet, der noch nicht gefixt wurde, allerdings gibt es einen Workaround, auf den später noch eingegangen wird

Nachfolgend soll nun das weitere Vorgehen dargelegt werden, mit dem diese Unterschiede beseitigt wurden und so eine Version des MyCompilers erstellt wurde, die diesen Use Case (parametrisiert) korrekt übersetzt.
Der erste hier aufgezählte Unterschied, ist auf einen Bug im MyCompiler zurück zuführen, der gefixt wurde (siehe Kapitel 5.2.2).
Die Lösung, des in Punkt zwei beschriebenen Unterschieds soll nun als nächstes dargelegt werden. Wie bereits beschrieben werden die Descriptoren bei Typparametern nicht auf "`Ljava/lang/Object;`" gesetzt sondern enthalten den Namen des Typparameters. So wird beispielsweise für die set – Methode (Zeile 10 in Listing 9) "`(LA;)V`" zurückgegeben, was soviel bedeutet, als dass, der Methode eine Instanz der 'lokalen' Klasse A übergeben wird. Wir wollen hier aber einen beliebigen Typ übergeben, außerdem gibt es die Klasse A nicht (bzw. es könnte sie nicht geben). Würde es sich allerdings an dieser Stelle um eine 'normale' Klasse handeln, also eine, die keine Typparameter enthält, wäre der erzeugte Descriptor richtig (es ist ja durchaus denkbar, dass eine Methode einen Parameter des Typs / der Klasse A übergeben bekommt).
Wie schon in Kapitel 5.2.2 (Fix des Object- und byte-Bugs) erkannt wurde, dient die Klasse `JVMCode` als zentrale Schnittstelle für die Descriptorengenerierung. Hier müsste nun erkannt werden, ob es sich um einen 'normalen' Typ oder um einen Typparameter handelt. Dies wurde dadurch erreicht, dass zusätzlich der Vector `paralist` (welcher ein Attribut der Klasse Class ist) mit übergeben wurde. Dieser Vector enthält alle Typparameter der zu compilierenden Klasse. Bevor die Methode `get_codegen_Type()` der Klasse `JVMCode` den Typ für den Descriptor zurückgibt muss sie prüfen, ob dieser Typ in dem Vector `paralist` enthalten ist. Ist dies der Fall, handelt es sich um einen Typparameter und es muss "`Ljava/lang/Object;`" zurückgegeben werden. Im gegensätzlichen Fall, dass der Typ in `paralist` nicht enthalten ist, kann die weitere Codegenerierung wie zuvor ablaufen.

[2] Die drei bis jetzt angesprochenen Use Cases weisen keinen Konstruktor auf, aus Testzwecken wurde jeder der drei Use Cases einmal mit und einmal ohne Konstruktor angelegt und dann mit allen sechs Version alle drei Compiler (sofern möglich) getestet.

Der dritte zuvor beschriebene Unterschied bestand in einem fehlenden aload_0 in den Konstruktoren und in der set- Methode des Use Cases. Um die Codeerzeugung für den Use Case fertig zustellen wurde in die Klasse `ClassFile` ein Workaround integriert, der das aload_0 einfügt. Dieser Workaround kann über dass Attribut `hamaAload0` de- bzw. aktiviert werden (`hamaAload0` ist standardmäßig auf false → einfügen des aload_0 deaktiviert).

5.4 Compilation der parametrisierten Klasse StoreSomethingPar

Wenn der zuvor beschriebene Workaround in der Klasse ClassFile aktiviert ist, wird die Klasse StoreSomethingPar von dem MyCompiler korrekt übersetzt. Um dies praktisch zu testen wurde eine Klasse StoreSomethingMain und StoreSomethingMainTrick geschrieben, die im Anhang D und im hama Verzeichnis enthalten sind. StoreSomethingMain wäre die korrekte Main – Klasse für den Use Case, allerdings kann der MyCompiler diese Datei nicht übersetzen, da die erforderlichen Funktionen noch nicht implementiert sind, der Sun Compiler kann dies ebenso wenig, da eine Instanz einer parametrisierten Klassen erzeugt wird. Aus diesem Grund wurde StoreSomethingMainTrick geschrieben, diese Klasse ist mit StoreSomethingMain identisch, außer das bei der Deklaration der übergebene Typ weggelassen wurde. Bei der Ausführung der Klasse StoreSomethingMainTrick ergibt sich folgende Ausgabe:

```
I'm a String
new String
I'm var b
new b
```

6 Erkenntnisse

Die Arbeit an dieser Studienarbeit brachte für mich in vielen Bereichen tiefere Kenntnisse. So wurde das Java Classfile Format sowie der Java Bytecode inklusive Opcodes und Mnemonics eingehend studiert. Außerdem gelang es mir im Umgang mit den verwendeten Werkzeugen eine größere Handfertigkeit zu erlangen. Dies betrifft hauptsächlich CVS und Eclipse. Ein Herausforderung bot die Einarbeitung in ein Projekt solcher Größe und das Management zwischen dieser Studienarbeit und der von Herr Ott.
Im nachhinein betrachtet hätte die Ausarbeitung der Dokumentation etwas früher beginnen sollen. Dies hätte auch zur Folge gehabt, dass zum Zeitpunkt der Rücksprache mit dem Betreuer ein Feedback zu den wesentlichen Teilen möglich gewesen wäre, was so mangels Ausarbeitung nicht möglich war.

7 Zusammenfassung

Fehlersuche und das Debugging sind trotz guter Hilfsmittel sehr zeitintensiv! Vielleicht der Hauptgrund, warum die Aufgabenstellung meinerseits nicht ganz erfüllt wurde, sind die kleinen Bugs, über die während des Projekts gestolpert wurde. Diese zogen meist eine Suche nach einem andern Use Case hinterher um die Bugsuche und das bugfixen zu vermeiden was manchmal gelang. In dem anderen Fall musste die betreffende Stelle im Sourcecode lokalisiert werden und dann Kontext abhängig eine Lösung gefunden werden.

8 Ausblick

Da das Projekt im Sinne dieser und der Studienarbeit von Herrn Ott nicht ganz abgeschlossen ist, wie es diesen Herbst voraussichtlich im Rahmen einer neuen Studienarbeit an einen weitern Studenten übergeben.

9 Literatur- Quellen- und Zitatverzeichnis

- [ADO04] Adobe,
  http://www.adobe.de/, Abruf am 17.01.2004

- [ECL03] eclipse,
  eclipse.org,
  http://www.eclipse.org/, Abruf am 17.09.2003

- [FRE04] Free Software Foundation, Inc.,
  GNU Make,
  http://www.gnu.org/software/make/, Abruf am 09.03.2004

- [HEL04] Helms Kai,
  CVS-Kurzanleitung,
  http://www.numerik.uni-kiel.de/~khe/cvs/cvs.html, Abruf am 15.03.2004

- [KOU03] Pavel Kouznetsov,
  Jad - the fast JAva Decompiler,
  http://kpdus.tripod.com/jad.html, Abruf am 17.11.2003

- [KÜH04a] Bernd Kühl (bekuehl@uos.de),
  JLex ein Scanner-Generator für Java,
  http://www.informatik.uni-osnabrueck.de/alumni/bernd/jay/staff/jlex/de/jlex.pdf,
  Abruf am 03.03.2004

- [KÜH04b] Bernd Kühl (bekuehl@uos.de),
  jay – Ein yacc für Java,
  http://www.informatik.uni-osnabrueck.de/alumni/bernd/jay/staff/design/de/design.pdf,
  Abruf am 07.03.2004

- [MAY03] Mayon Software Research,
  ClassCracker 3 Java decompiler,
  http://mayon.actewagl.net.au/, Abruf am 28.10.2003

- [MID04] MID GmbH,
  INNOVATOR - die durchgängige Standardsoftware für objekt- und funktionsorientierte Software-Entwicklung sowie Geschäftsprozess- und Datenmodellierung
  http://www.mid.de/de/innovator/, Abruf am 21.02.2004

- [OMO04] Omondo,
  Omondo EclipseUML,
  http://www.omondo.de/, Abruf am 22.01.2004

- [OTT04] Ott Thomas,
  Studienarbeit: Typinferenz in Java,
  Berufsakademie Horb, Juni 2004

- [PIZ03] The Pizza Compiler,
  The Pizza Compiler, an Open Source compiler for a Java superset,
  http://pizzacompiler.sourceforge.net/, Abruf am 20.09.2003

- [PLÜ03] Dr. Prof. Martin Plümicke,
  jvmDisassembler

- [REI03] Reichenbach Felix,
  Studienarbeit: Erweiterung der semantischen Analyse des Java – Compilers,
  Berufsakademie Horb, Mai 2003

- [SUN04a] Sun Microsystems,
  Java Technology,
  http://java.sun.com, Abruf am 07.02.2004

- [SUN04b] Sun Microsystems,
  Java Technology, Java 2 Platform, Standard Edition (J2SE),
  http://java.sun.com/j2se/index.jsp, Abruf am 07.02.2004

- [SUN04c] Sun Microsystems,
  Java Technology, Java 2 Platform, Enterprise Edition (J2EE),
  http://java.sun.com/j2ee/index.jsp, Abruf am 07.02.2004

- [SUN04d] Sun Microsystems,
  Java Technology, Java 2 Platform, Micro Edition (J2ME),
  http://java.sun.com/j2me/index.jsp, Abruf am 07.02.2004

- [SUN04e] Sun Microsystems,
  Servlets and JSP Pages Best Practices,
  http://java.sun.com/developer/technicalArticles/javaserverpages/servlets_jsp/

- Die virtuelle Maschine von Java
  http://www.num.math.uni-goettingen.de/Lehre/Lehrmaterial/Vorlesungen/Informatik/1998/skript/texte/VM.html (22.04.2004)

- Java Byte Code & Class Files (Christof Senn)
  http://www.cs.fh-aargau.ch/~gruntz/courses/sem/ws03/JavaClassFiles.pdf (05.02.2004), S. 16

- Java 2 Platform, Standard Edition, v 1.3.1 API Specification
  http://java.sun.com/j2se/1.3/docs/api/overview-summary.html (17.03.2004)

- Java Magazin
  http://javamagazin.de/ (24.01.2004)

- Java User Group Deutschland
  http://www.java.de/ (02.02.2004)

- JDBC 3.0 Specification (Sun Microsystems, Inc.)
  ftp://ftp.javasoft.com/pub/jdbc/dwh145674/jdbc-3_0-pfd4-spec.pdf (06.02.2004), S. 25

- Telekom Netzwerk-Lexikon
  http://www.t-lan.de/glossar/glossar.asp (15.03.2004)

- The Java Virtual Machine Specification
  http://java.sun.com/docs/books/vmspec/index.html (26.01.2004)

- The Java Virtual Machine Specification, Second Edition (Tim Lindholm, Frank Yellin)
  http://java.sun.com/docs/books/vmspec/2nd-edition/html/VMSpecTOC.doc.html (29.01.2004)

- IT - spezifische Enzyklopädie
  http://www.whatis.com/ (14.02.2004)

- Orientation In Objects, Warum mit Eclipse entwickeln,
  http://www.oio.de/public/warum-eclipse.htm (22.03.2004)

- Unified Modeling Language, UML Resource Page,
  http://www.uml.org/ (17.10.2003)

- GoTo Java 2, 2. Auflage (Guido Krüger)
  Addison-Wesley, 2000, S. 34

- Java 2 in 21 Tagen (Laura Lemay, Rogers Cadenhead)
  Markt & Technik, 1999, S.18

- Java Virtual Machine, Sprache Konzept Architektur (Dallheimer, Kalle)
  O'Reilly Verlag, 1997, S. 55

- Übersetzung objektorientierter Programmiersprachen (Bauer/Höllerer)
  Springer Verlag, Oktober 1998, ISBN: 3540642560, S. 67

- The Java Virtual Machine Specification, Second Edition (Tim Lindhol, Frank Yellin)
  Addison Wesley, Februar 1999, ISBN: 0201432943, S. 134ff

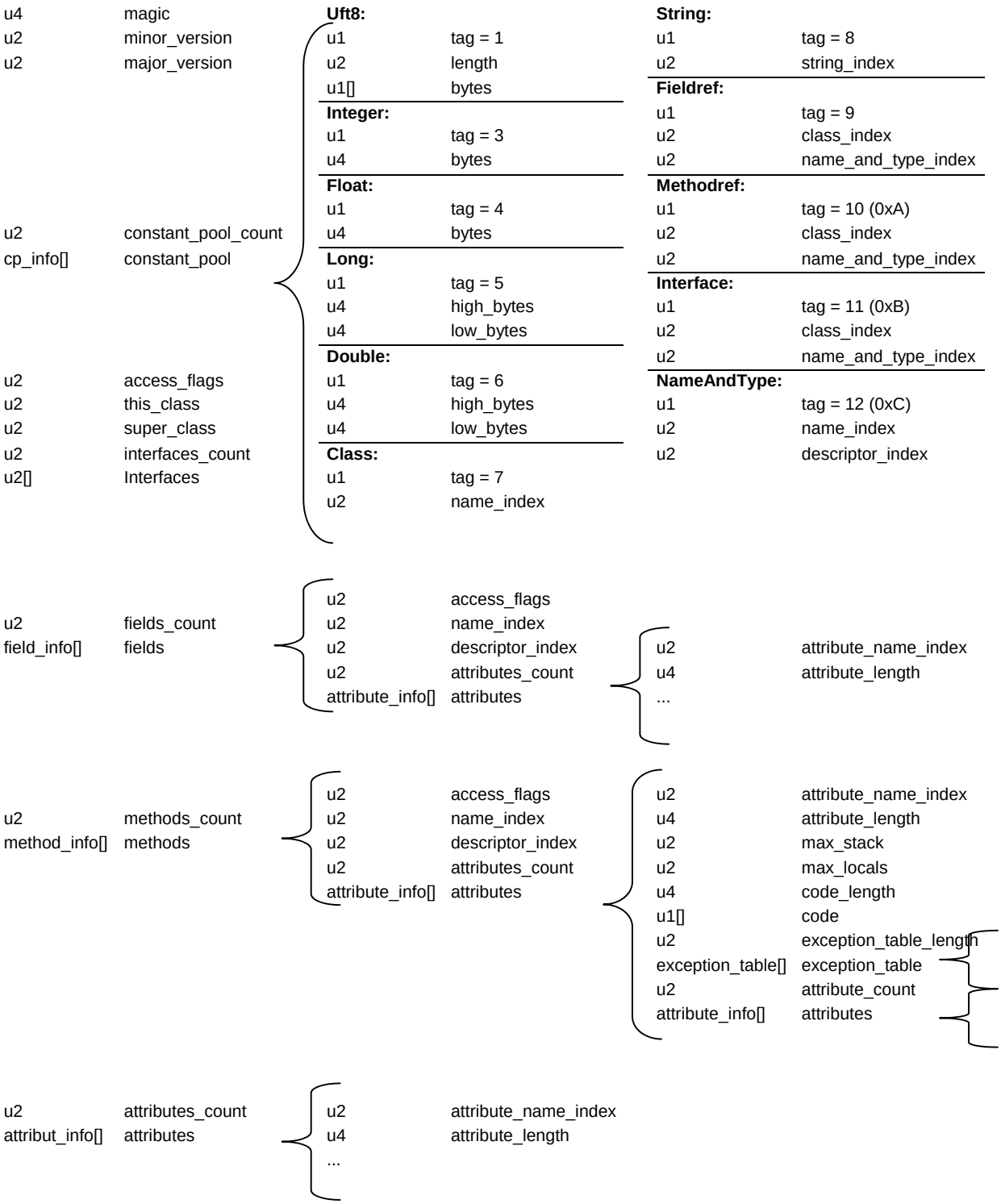
10 Abbildungsverzeichnis

11 Tabellenverzeichnis

12 Listingverzeichnis

13 Anhang

A) Das Classfile Format auf einen Blick



14 Anhang

B) Use Case Empty, Add und Ergebnis

Datei /hama/Empty.jav

```
class Empty
{

}
```

Datei /hama/Add.jav

```
class Add
{
     int add(int a, int b)
     {
          return a + b;
     }
}
```

Datei /hama/Ergebnis.jav

```
class Ergebnis
{
     int zahl1;
     int zahl2;
     int erg;

     void berechne(int a, int b)
     {
          zahl1 = a;
          zahl2 = b;
          erg = zahl1 + zahl2;
     }

     int ergebnis()
     {
          return erg;
     }
}
```

14 Anhang

C) Use Case Setze und Toggle1

Datei /hama/Setze.jav

```
class Setze
{
     int zahl;

     void setze()
     {
          zahl = 555;
     }
}
```

Datei /hama/Toggle1.jav

```
class Toggle1
{
     boolean toggle(boolean b)
     {
          if(b)
          {
               b = false;
               return b;
          }
          else
          {
               b = true;
               return b;
          }
     }
}
```

14 Anhang

D) Die Main Klassen **StoreSomethingMain und StoreSomethingMainTrick**

Datei /hama/StoreSomethingMain.jav

```
public class StoreSomethingParMain
{
     public static void main(String[] args)
     {
          StoreSomethingParCon<String> a = new
StoreSomethingParCon("I'm a String");
          System.out.println(a.get());
          a.set("new String");
          System.out.println(a.get());
          StoreSomethingPar<String> b = new
StoreSomethingPar();
          b.set("I'm var b");
          System.out.println(b.get());
          b.set("new b");
          System.out.println(b.get());
     }
}
```

Datei /hama/ StoreSomethingMainTrick.jav

```
public class StoreSomethingParMainTrick
{
     public static void main(String[] args)
     {
          StoreSomethingParCon a = new
StoreSomethingParCon("I'm a String");
          System.out.println(a.get());
          a.set("new String");
          System.out.println(a.get());
          StoreSomethingPar b = new StoreSomethingPar();
          b.set("I'm var b");
          System.out.println(b.get());
          b.set("new b");
          System.out.println(b.get());
     }
}
```