Duale Hochschule Baden-Württemberg Mannheim

Studienarbeit

Erweiterung des Java-TX-Compilers um Pattern Matching im Stil funktionaler Programmiersprachen

Studiengang Informatik

Studienrichtung Informationstechnik

Verfasser(in):	Luca Trumpfheller
Matrikelnummer:	7297551
Kurs:	TINF20IT2
Studiengangsleiter:	Prof. Dr. Nathan Sudermann-Merx
Wissenschaftliche Betreuer:	Prof. Dr. habil. Martin Plümicke, Andreas Stadelmeier
Bearbeitungszeitraum:	09.05.2022–19.07.2023
Eingereicht:	19.07.2023

Ehrenwörtliche Erklärung

Ich versichere hiermit, dass ich die vorliegende Arbeit mit dem Titel "*Erweiterung des Java-TX-Compilers um Pattern Matching im Stil funktionaler Programmiersprachen*" selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe. Ich versichere zudem, dass die eingereichte elektronische Fassung mit der gedruckten Fassung übereinstimmt.

Mannheim, 18.07.2023

Ort, Datum

Luca Trumpfheller

Inhaltsverzeichnis

Abbildungsverzeichnis

Quelltextverzeichnis

Abkürzungsverzeichnis

AST	Abstract Syntax Tree
DHBW	Duale Hochschule Baden-Württemberg
JDK	Java Development Kit
JEP	JDK Enhancement Proposal
JVM	Java Virtual Machine
LTS	Long Term Support
TX	Type Extended

1 Einleitung

Das Fachgebiet Compilerbau zählt zu den am besten erforschten Teilbereichen der Informatik [39]. Durch die Entwicklung der Informatik selbst, sowie den Einzug digitaler Technologien in viele andere Industriezweige, ergeben sich ständig neue Anforderungen an Programmiersprachen. Entweder werden neue Sprachen, und damit Compiler, entwickelt oder bestehende Sprachen werden um neue Funktionen ergänzt. Vor allem wird in regelmäßigen Abständen eine höhere Effizienz bei der Ausführung erwartet. Deshalb kann man einen Compiler bzw. eine Programmiersprache grundsätzlich als dynamisches System betrachten, das sich in einer ständigen Evolution befindet. Als populäres Beispiel hierfür dient die Programmiersprache Java. Seit der Veröffentlichung der ersten Java Version 1996 durch "Sun Microsystems" wurden jährlich mehrere Updates veröffentlicht [3]. Mittlerweile ist Java Version 20 auf dem Markt und erfreut sich weiterhin großer Beliebtheit [19]. Unabhängigkeit vom Plattformen, eine große Vielzahl an Funktionen und effiziente, schnelle Programmlaufzeiten tragen zum Erfolg und zur ständigen, erfolgreichen Weiterentwicklung der Programmiersprache bei. Neben Updates für Bugs und Sicherheitslücken werden auch regelmäßig neue Konzepte aus der Theoretischen Informatik und dem Compilerbau integriert. Eine populäres Beispiel hierfür ist die Einführung von generischen Typen ("Generics") in Java Version 5 (1.5) [16].

Auch an der Duale Hochschule Baden-Württemberg (DHBW) trägt man zur Weiterentwicklung von Java bei. Am Standort Horb der DHBW wird seit rund 10 Jahren an einer Erweiterung der Programmiersprache Java geforscht. Im Rahmen des Forschungsschwerpunkts "Typsysteme für objektorientierte Programmiersprachen" wird der "Java TX"-Compiler entwickelt. Dieser Compiler ergänzt die Programmiersprache Java um globale Typinferenz und ihr Typsystem um echte Funktionstypen [37] [49].

1.1 Motivation

Die Sprache "Java TX" erweitert die Programmiersprache Java und möchte so zu deren Entwicklung beitragen. Da die Funktionen für globale Typinferenz und echte Funktionstypen bis heute nicht in Java implementiert wurden, möchte man deren Einsatz mit "Java TX" erforschen. Es existieren bereits Programmiersprachen, in denen diese Konzepte bereits

implementiert sind. Dabei handelt es sich oft um sogenannte funktionale Programmiersprachen. Seit Java Version 1.5 wurden auch bereits einige Funktionen funktionaler Programmiersprachen in Java implementiert. Diese Entwicklung soll durch "Java TX" fortgeführt werden. Der "Java TX"-Compiler ist in Java implementiert und enthält einen selbstentwickelten Algorithmus für globale Typinferenz. Außerdem unterstützt er echte Funktionstypen und arbeitet mit allgemeine Typen (engl. "Principal Types", siehe Kapitel 2.5 auf Seite 24) [49].

Diese Erweiterungen der Programmiersprache Java sollen die Arbeit der Programmierer einfacher und effizienter gestalten [37]. Außerdem wird die Programmiersprache Java in "Java TX" um Funktionen erweitert, die aus funktionalen Programmiersprachen, wie zum Beispiel "Haskell", bekannt sind und seit einiger Zeit dort eingesetzt werden [20] [36] [23]. So können Vorteile objektorientierter Sprachen mit den Vorteilen funktionaler Programmiersprachen verknüpft werden und bieten Anwendern weitere Möglichkeiten zur Problemlösung. Gleichzeitig wurden, durch die Erweiterungen, Nachteile in Java Version 8, auf der "Java TX" basiert, eliminiert [48].

1.2 Problemstellung

Seit der Veröffenlichung der Java Version 8 im März 2014 hat sich die Programmiersprache ständig weiterentwickelt [3]. Es wurden zahlreiche neue Funktionen in verschiedenen Releases eingeführt. Der "Java TX"-Compiler ist jedoch noch auf dem Stand der Java Version 8. Das bedeutet, dass der "Java TX"-Compiler neue Java-Funktionen nicht verarbeiten kann und Entwicklern einige Sprachkonstrukte vorenthält, die in Java bereits zur Verfügung stehen. Im Rahmen dieser Arbeit sollen nun erste Schritte unternommen werden, um "Java TX" zu aktualisieren und verschiedene aktuelle Funktionen und Sprachkonstrukte zu integrieren. Dazu gehören beispielsweise "Pattern Matching", "Records" oder "Sealed Classes". So soll gewährleistet werden, dass "Java TX" auch aktuelle Java Versionen erweitert und nicht allzu weit hinter der Entwicklung von Java bleibt.

Besondere Bedeutung für den Forschungsschwerpunkt "Typsysteme für objektorientierte Programmiersprachen" hat dabei die Einführung des "Pattern Matchings", das seinen Ursprung in funktionalen Programmiersprachen hat [21]. Seine Integration in Java begann in zwei Formen in den Versionen 14 bis 19 und dauert bis heute an. Zunächst wurde das Konzept des "Pattern Matching" durch den neuen "instanceof"-Operator in den Versionen 14 bis

16 eingeführt [29] [30] [31]. Anschließend begann die Erweiterung des "switch"-Statements zur Unterstützung von "Pattern Matching" in den Versionen 17 bis 19 [10] [11] [12]. Das Potential für "Pattern Matching" in "Java TX" ist bereits seit 2019 bekannt und soll nun ausgeschöpft werden [49]. Eine detaillierte Beschreibung des Konzepts und seiner Funktionsweise folgt in Kapitel 2.6.1 auf Seite 36.

Zusätzlich sollen zwei neue Sprachkonstrukte aus Java, "Sealed Classes" und "Records", in "Java TX" aufgenommen werden. Beide Features stehen in Verbindung mit dem "Pattern Matching" und tragen zum Ziel der Vereinfachung der Programmierarbeit mit "Java TX" bei [34] [33]. Die Entwicklung der sogenannten "Records" findet parallel zum "Pattern Matching" in "switch"-Statements statt und dauert noch an [14]. Die Implementierung der "Sealed Classes" fand in Java Version 17 ihren Abschluss und ist seitdem Bestandteil der Programmiersprache [34].

1.3 Ziel der Arbeit

Ziel ist es, sämtliche Komponenten, die an der Kompilierung von "Java TX"-Programmen beteiligt sind, zu erweitern. Sie sollen alle genannten Funktionen und Sprachkonstrukte verarbeiten können, so dass "Pattern Matching", "Records" und "Sealed Classes" auch in "Java TX"-Programmen verwendet werden können. Dazu werden zunächst im folgenden Kapitel 2 auf der nächsten Seite einige theoretische Grundlagen zu Compilern und "Java TX" erarbeitet. Außerdem wird die Entwicklung der Programmiersprache Java seit Version 8, in Bezug auf die genannten Funktionen und Sprachkonstrukte, untersucht. Anschließend wird der Entwicklungsprozess zur Aktualisierung des "Java TX"-Compilers dargestellt (siehe Kapitel 3 auf Seite 48). Folgende Implementierungsschritte sollen vorgenommen werden:

1. Erweiterung des AST des Compilers zur Abbildung neuer Sprachkonstrukte
2. Erstellen von "Unit-Tests" für den aktualisierten Generator für ASTs
3. Anpassung des Algorithmus für globale Typinferenz an verändertem AST

2 Theoretische Grundlagen

Das folgende Kapitel beschreibt relevante theoretische Grundlagen für das Verständnis der Entwicklungstätigkeit zur Erweiterung des "Java TX"-Compilers um neue Java Features. Zunächst werden Grundlagen eines Compilers erläutert. Dazu zählen Prozesse wie die Syntaxanalyse oder die Generierung von Maschinencode sowie verwendete Datenstrukturen wie ASTs. Dabei wird speziell auf die Programmiersprache Java eingegangen, da diese Sprache, beziehungsweise der zugehörige Compiler, die Grundlage für den "Java TX"-Compiler darstellt.

Danach folgt eine Beschreibung der Funktionen in "Java TX", welche die Programmiersprache Java erweitern. Anschließend wird die Entwicklung der Programmiersprache Java seit Version 8 untersucht. Dabei werden besonders solche Neuerungen berücksichtigt, die im Zusammenhang mit dem sogenannten "Pattern Matching" stehen. Dieses Konzept soll, zusammen mit weiteren neuen Funktionen, im Rahmen dieser Arbeit in "Java TX" eingeführt werden.

2.1 Compiler

Ein Computer, beziehungsweise der in ihm enthaltene Prozessor, arbeitet mit Anweisungen, die durch bestimmte Folgen von Bytes dargestellt werden. Eine solche Sprache, die Abfolgen von Zahlen verwendet, ist für Entwickler, also Menschen, nicht geeignet. Die direkte Entwicklung von Software in Maschinencode wäre höchst ineffizient, da Menschen solchen Code recht schwer verarbeiten und schreiben können. Um die Komplexität der Maschinensprache zu abstrahieren und den Softwareentwicklungsprozess effizienter zu gestalten, wurden Programmiersprachen und zugehörige Compiler entwickelt [5]. Als Compiler wird ein Programm bezeichnet, welches im Rahmen der Softwareentwicklung auf Computern ausgeführt wird. Die Aufgabe eines Compilers ist die Übersetzung von menschenlesbarem Quelltext zu ausführbarem Maschinencode. Dieser Prozess wird als "Kompilierung" bezeichnet. Während dieser Umwandlung zwischen den beiden Sprachen prüft ein Compiler außerdem, ob die Eingabe frei von Fehlern ist und führt Optimierungen durch. Dieser spezielle Übersetzer spielt bei der Entwicklung von Computerprogrammen eine wichtige Rolle, da er die Komplexität der Maschinencodes abstrahiert und es Entwicklern ermöglicht, leichter

verständliche Programmiersprachen, die natürlichen Sprachen ähneln, in Maschinencode umzuwandeln. Je nach verwendeter Programmiersprache und Computerarchitektur unterscheiden sich Compiler in ihrer Funktionsweise und ihren Übersetzungsfunktionen beziehungsweise -vorgängen. Jedoch verfolgen sämtliche Compiler dasselbe Ziel. Daher wurde im Rahmen der Entwicklung und Forschung zum Compilerbau ein allgemeiner Kompilierungsprozess formuliert. Dieser komplexe Prozess kann auf dem höchsten Abstraktionsniveau in zwei Phasen aufgeteilt werden. Man spricht dabei vom "Frontend" und "Backend" des Compilers [39].

Frontend Das "Frontend" operiert auf dem Quelltext, der in einer Hochsprache — wie C, Java, etc. — geschrieben ist und ein Programm repräsentiert. Das "Frontend" prüft den Quelltext in der Analysephase auf formale Korrektheit. Ist das Programm formal korrekt, erzeugt ein Compiler eine Zwischendarstellung des Programms, die als Grundlage für die vom "Backend" auszuführende Synthesephase dient. Diese Zwischendarstellung enthält nur Informationen, die für den weiteren Ablauf des Kompiliervorganges notwendig sind. Diese Informationen, über im Programm enthaltenen Strukturen, werden während dessen Analyse gesammelt und in geeigneten Datenstrukturen, wie Listen, Tabellen und ASTs (vgl. 2.4.1 auf Seite 22), für die Zwischendarstellung aufbereitet. Typischerweise wird ein Programm in drei Teilphasen analysiert. Ein "Scanner" erkennt den lexikalischen Aufbau eines Programms und generiert "primitive, syntaktische Elemente" [39], auch "Tokens" genannt. Beispiele für solchen "Tokens" sind sämtliche Namen, die in einem Computerprogramm zur Identifikation von Sprachkonstrukten (z.B. Variablen, Methoden, Klassen, etc.) verwendet werden (Englisch: "identifier"), syntaktische Elemente wie eckige Klammern ("[", "]") oder bestimmte Funktionswörter, die Kontrollstrukturen definieren (Englisch: "keywords"), wie "if", "for", "else" oder "while". Dadurch stellt ein Compiler sicher, dass im Quelltext nur erlaubte Zeichen beziehungsweise Zeichenfolgen verwendet wurden und der "Parser" (vgl. Abschnitt 2.2.2 auf Seite 13) ausschließlich relevante "Tokens" zur Verarbeitung erhält. Ein Beispiel für einen irrelevanten "Token" ist das Leerzeichen. Es hat keinen Einfluss auf die Ausführung eines Computerprogrammes und dient meist ausschließlich des Lesbarkeit des Quelltextes. Deshalb wird es im Rahmen der lexikalischen Analyse aus der Zeichenfolge einer Quelldatei entfernt. Die relevanten Tokens werden von einem "Parser" dazu verwendet, um die syntaktische Korrektheit des Quellcodes zu überprüfen und die Hierarchie eines Programms zu analysieren. Die Struktur einer Programmiersprache ist durch eine kontextfreie Grammatik definiert [5]. Diese wird vom Parser verwendet, um sicherzustellen, dass die Entwickler die Syntax einer Programmiersprache gemäß ihrer

Spezifikation (z.B. Spezifikation der Java Version 8 [54]) in ihrem Quelltext einhalten.

Definition 1 *Syntax: Der Begriff Syntax beschreibt eine Sammlung von Vorgaben für die Kombination elementarer Zeichen zu zusammengesetzten Zeichen. Im Kontext von Programmiersprachen gibt die Syntax Regeln für korrekte Programmtexte, die aus einem definierten Alphabet gebildet werden, vor [26].*

Außerdem wird die Semantik des Codes im "Frontend" analysiert, um den Einsatz von Typen zu prüfen und gegebenenfalls veranlasste Typumwandlungen zu berechnen [39]. Die semantische Analyse des Java-Compilers wird in "Java TX" beispielsweise um Typinferenz erweitert (vgl. 2.5.2 auf Seite 27). Die drei Teilaspekte eines Programms werden während dessen Kompilierung zwar durch einzelne Funktionen untersucht, diese Funktionen laufen jedoch nicht unbedingt prozedural ab. Natürlich unterscheiden sich verschiedene Compiler in der Implementierung dieser Funktionen. Im Falle des in dieser Arbeit behandelten Compilers interagieren die analytischen Komponenten miteinander und untersuchen den Eingabetext gemeinsam. So ruft der "Parser" zum Beispiel den "Scanner" auf, um den nächsten "Token" des Quelltextes für seine syntaktische Analyse zu erhalten. Dies erspart eine doppelte Iteration durch den Eingabetext.

Backend Auf Basis der Zwischendarstellung kann der Compiler, in seinem sogenannten "Backend", in der Synthesephase Maschinencode generieren, der von Computern direkt oder über eine virtuelle Umgebung (vgl. Java Virtual Machine (JVM) [43]) ausgeführt werden kann. Diese Umwandlung ist plattform- und sprachspezifisch. Klassische Beispiele für Maschinencode sind die Befehlssätze für Intel's i386- und die MIPS-Architekturen[17]. Einige Compiler unterteilen die Synthesephase noch. So kann beispielsweise generischer Zwischencode erzeugt werden. Dieser Code befindet sich auf einem Abstraktionsniveau zwischen der Hochsprache, die von Entwicklern im Quelltext verwendet wird, und dem Maschinencode, der vom Compiler für die jeweilige Prozessorarchitektur erzeugt wird. Art und Aufbau dieses Codes können von den Herausgebern einer Programmiersprache bestimmt werden. Er kann, unabhängig vom Zielsystem, optimiert werden. So können beispielsweise Rekursionen entfernt oder Folgen von Sprüngen zwischen Instruktionen verkürzt werden. Der Zwischencode bleibt in der Regel vor den Anwendern einer Programmiersprache verborgen. Durch dessen Verwendung wird die Entwicklung von plattformunabhänigen Programmiersprachen vereinfacht, da nur der finale Schritt der Synthesephase, die Generierung von Maschinencode, abhängig von der verwendeten Computerarchitektur ist [39].

Im Kontext dieser Arbeit ist als Beispiel für einen Zwischencode der Java "Bytecode" zu nennen. Dieser Zwischencode wird vom Java-Compiler generiert und von der JVM zur Laufzeit in den passenden Maschinencode für die ausführende Plattform umgewandelt [43] (siehe auch folgender Abschnitt). Dieser Mix aus Kompilieren und Interpretieren einer Programmiersprache stellt eine Besonderheit dar und macht Java äußerst attraktiv für die Verwendung für plattformunabhängige Software.

2.1.1 Kompilierung und Ausführung von Java-Programmen

Als plattformunabhängige Programmiersprache setzt Java einen zweiphasigen Prozess zur Übersetzung von Quelltext in Maschinencode ein. Damit unterscheidet sie sich von Programmiersprachen wie C, deren Kompilierungsprozess in einer einzigen Phase abläuft. Das "Backend" des C-Compilers generiert Maschinencode für die zugrundeliegende Hardwarearchitektur. Für jede Architektur existiert also eine eigene Version des C-Compilers [28]. So entsteht eine Abhängigkeit von den einzelnen Plattformen und ein Programm, das auf einem Computer mit Intel-Prozessor kompiliert wurde, kann nicht auf einem Server mit PowerPC-Architektur ausgeführt werden.

Java verfolgt einen anderen Ansatz und setzt für die Umwandlung von Quelltext zu Maschinencode zwei verschiedene Komponenten ein. Durch den Einsatz einer virtuellen Maschine (der JVM) entfällt die Notwendigkeit für die Umwandlung von Java-Programmen in Maschinencode während der Kompilierung. Der "Bytecode" mit dem die JVM arbeitet, ist unabhängig von Plattformen und bietet Entwicklern Portabilität für ihre Anwendungen. Erst bei der Ausführung der Programme in der JVM wird Maschinencode für die entsprechende Hardwarearchitektur generiert [17]. Es ist zu beachten, dass, je nach Betrachtungsweise, auch der Java "Bytecode" als Maschinencode bezeichnet werden kann, da er in der JVM ausgeführt wird. Diese virtuelle Maschine weist, wie Hardwareplattformen, eine klar definierte Architektur und einen Befehlssatz auf [18]. Betrachtet man jedoch den gesamten Lebenszyklus eines Java-Programms, inklusive dessen Ausführung auf der Hardware eines Computers, stellt der Java "Bytecode" eher einen Zwischencode dar, der während seiner Laufzeit in Maschinencode für die passende Architektur umgewandelt wird [22].

Der Java-Compiler ist wie ein gewöhnlicher Compiler aufgebaut und kompiliert Quelltext in "Front-" und "Backend". Jedoch handelt es sich beim generierten Code des Java-Compilers nicht um Maschinencode zur direkten Ausführung auf Hardware. Der Compiler erstellt eine

Repräsentation eines Programms in einem Zwischencode, beziehungsweise im Maschinencode der JVM, dem sogenannten "Bytecode". Dazu werden alle angegebenen Quelldateien vom "Frontend" des Compilers analysiert. Dabei wird eine Zwischendarstellung, bestehend aus Symboltabelle und AST, erstellt und semantisch analysiert. Außerdem wird der sogenannte "Syntactic Sugar" aus dem AST entfernt (mehr dazu im Abschnitt 2.3 auf Seite 19). Abschließend wandelt der Compiler die Zwischendarstellung des Programms in "Bytecode" um. Dieser wird in Dateien mit der Endung ".class" gespeichert und kann von der JVM ausgeführt, beziehungsweise in Maschinencode für die passende Plattform umgewandelt werden [17] [22].

```
1  public java.lang.Object();
2      Code:
3         0: return
4
5    public final native java.lang.Class<?> getClass();
6
7    public native int hashCode();
8
9    public boolean equals(java.lang.Object);
10     Code:
11        0: aload_0
12        1: aload_1
13        2: if_acmpne 9
14        5: iconst_1
15        6: goto 10
16        9: iconst_0
17        10: ireturn
18 [...]
```

Quelltext 2.1: Beispiel für Java "Bytecode": Auszug aus der Datei "Object.class" der Java-Standardbibliothek

Diese Trennung von Kompilierung und Ausführung von Java-Programmen schafft Unabhängigkeit von Hardwarearchitekturen. Jedoch muss die JVM an die jeweilige Plattform, auf der sie ausgeführt werden soll, um Java "Bytecode" in Maschinencode zu übersetzen, angepasst werden. Anpassungen und Erweiterungen des Java-Compilers werden wiederum vereinfacht, da Entwickler sich "nur" um die Generierung von "Bytecode" für die JVM

kümmern müssen. Die finale Übersetzung des "Bytecodes" in architekturspezifischen Maschinencode ist nicht Teil des Java Compilers. Dieser Umstand wird auch bei der Entwicklung von "Java TX" ausgenutzt. Der "Java TX"-Compiler erstellt "Bytecode", der mit der Spezifikation der JVM kompatibel ist. Dadurch ist keine Anpassung dieser virtuellen Umgebung notwendig und "Java TX"-Programme können auf sämtlichen Computern mit installierter JVM ausgeführt werden.

2.2 Analysephase eines Compilers

Im "Frontend" (vgl. Abschnitt 2.1 auf Seite 5) eines Compilers durchläuft ein Computerprogramm die Analysephase des Compilers (vgl. 2.1 auf Seite 5). Die beiden Komponenten "Scanner" und "Parser" nehmen während dieser Phase eine besondere Rolle. Sie arbeiten gemeinsam mit der Eingabe der Programmierer und erstellen die erste maschinenlesbare Repräsentation des Programms und sind für die Prüfung dessen Syntax verantwortlich. Dazu wird der Quelltext in Form einer Folge von "Tokens" eingelesen und mithilfe einer kontextfreien Grammatik auf Korrektheit überprüft. Das Zusammenspiel der beiden Komponenten resultiert in einem sogenannten "Parsebaum". Dieser Baum stellt das eingegeben Programm syntaktisch dar und kann gut vom Computer, der den Compiler ausführt, weiterverarbeitet werden — besser als der Quelltext der Entwickler.

Wie bereits erwähnt, arbeiten die beiden Komponenten, je nach Aufbau eines Compilers, auf verschiedene Weisen miteinander (vgl. Abschnitt 2.1 auf Seite 5). Es gibt Compiler, die "Scanner" und "Parser" nacheinander ausführen. Zunächst wird der Quelltext vom "Scanner" durchlaufen. Dieser teilt ihn in "Tokens" auf und gibt eine Folge solcher "Tokens" an den Compiler zurück. Wurden unzulässige Zeichen im Programm verwendet, schlägt die lexikalische Analyse fehl und der "Scanner" meldet einen Fehler im Kompiliervorgang. Anschließend wird der "Parser" aufgerufen, um die Syntax der "Tokenfolge" zu analysieren. Die "Tokens" werden ihm dafür in einer geeigneten Datenstruktur übergeben (beispielsweise in einem "Array" von "Strings"). Sowohl der Java als auch "Java TX"-Compiler verfolgen jedoch einen anderen Ansatz. Ziel ist die Analyse des Programms durch "Scanner" und "Parser" in einem Schritt durchzuführen. Dazu arbeiten "Scanner" und "Parser zusammen. Der "Parser" beginnt den Analyseprozess, indem er den ersten Token des Quelltextes vom "Scanner" anfordert. Die enthaltenen "Token" setzt der "Parser" zusammen und analysiert die resultierende Folge auf syntaktische Korrektheit (Details dazu folgen in Abschnitt 2.2.2

auf Seite 13). "Scanner" und "Parser" arbeiten bei der Analyse eines Programms eng zusammen. So kann Rechenaufwand und Zeit bei der Analyse eines Computerprogramms gespart werden. Compiler durchlaufen den Quelltext eines Programms nur einmal und arbeiten dadurch effizienter [55].

2.2.1 Scanner

Die Aufgabe des "Scanners" ist die Einteilung von Eingabetext, der ein Computerprogramm darstellt, in kleinstmögliche Einheiten, die vom Compiler (beziehungsweise dem "Parser") einfach weiterverarbeitet werden können. Grundsätzlich könnte man den Quelltext natürlich in einzelne Zeichen zerlegen. Diese Zeichenfolge wird dann durchlaufen und auf die Einhaltung der Syntax der Programmiersprache untersucht. Dazu müssen jedoch einzelne Zeichenkombinationen aus der Zeichenfolge extrahiert werden, um definierte Wörter für Kontrollstrukturen oder Sprachkonstrukte einer Programmiersprache herausfiltern zu können. Insgesamt wäre der Prozess der manuellen Analyse sämtlicher Einzelzeichen, vor allem für große Computerprogramme mit viel Quellcode, äußerst komplex und zeitaufwendig.

Der Prozess der Analyse der verwendeten lexikalischen Elemente, der kleinsten syntaktischen Einheiten, wird in modernen Compilern durch einen Scanner durchgeführt. Diese Komponente stellt dem "Parser" die einzelnen "Token" eines Quelltextes zur Verfügung. Diese "Token" werden durch die Anwendung regulärer Ausdrücke auf den Eingabetext aus diesem extrahiert. So können häufig verwendete Zeichenfolgen beziehungsweise Wörter erkannt werden und einem einheitlichen "Token" zugeordnet werden. Anhang dieses "Tokens" erkennt der "Parser" dann, dass der Entwickler an einer bestimmten Stelle des Quellcodes ein bestimmtes "Keyword" verwendet hat. Auch für die Erkennung von Namen für Deklarationen, sogenannte "Identifier", eignen sich reguläre Ausdrücke. Ein regulärer Ausdruck genügt, um die Beschränkungen für Variablennamen in Java zu überprüfen.

Der "Scanner" arbeitet also mit einer Liste von regulären Ausdrücken, die er wiederholt mit dem Quelltext abgleicht. Diesen Ausdrücken ist jeweils eine Aktion zugewiesen. Meist gibt der "Scanner" einen "Token", der die extrahierte Zeichenfolge klassifiziert, zurück. Er kann aber beispielsweise auch Zeichen verwerfen, wenn diese nur der Lesbarkeit des Quelltextes dienen und nicht weiter relevant für den Compiler sind. Wird der "Scanner" im Rahmen der Syntaxanalyse vom "Parser" aufgerufen, um einen "Token" aus dem Quelltext zu extrahieren, liest er zeichenweise Quellcode ein und wendet reguläre Ausdrücke auf diese

Zeichen an. Die Aktion, die mit dem regulären Ausdruck verknüpft ist, den die eingelesenen Zeichen zuerst "erfüllen", wird vom "Scanner" ausgeführt. Je nach Art des "Token" kann es sich bei der tatsächlich repräsentierten Zeichenfolge um ein einziges oder mehrere Zeichen handeln. Der "Token" `LBRACK` könnte beispielsweise das Zeichen "[" repräsentieren. Der "Token" `CONT` das "Keyword" `continue` für das Starten des nächsten Schleifendurchlaufs. Die Art des "Tokens" und die repräsentierten Zeichen werden für die Analysephase zusammen gespeichert. Im Falle des "Java TX"-Compilers handelt es sich dabei um Objekte der Klasse "Token", die dann vom "Parser" weiterverwendet werden (siehe folgender Abschnitt 2.2.2 auf Seite 13).

Eine Besonderheit des "Scanners" ist, dass er die korrekte Vergabe von Name für Variablen, Methoden oder Klassen zu Beginn der Analysephase überprüfen kann. Durch einen regulären Ausdruck können die Kriterien für einen korrekten Variablen-, Funktions- oder Klassennamen vollständig definiert werden. In Java müssen Variablennamen beispielsweise mit einem Buchstaben, Dollarzeichen oder Unterstrich beginnen und dürfen, ausgenommen von Zahlen, keine anderen Zeichen enthalten [54]. Die Regeln für Methoden- und Klassennamen können bei Bedarf durch weitere "Token" geprüft werden. Die "Token" für sogenannte "Identifier" für Sprachkonstrukte wie Variablen, Methoden und Klassen stehen meist am Ende der "Token-Liste". Das bedeutet, dass sie erst in Betracht gezogen werden, wenn kein anderer "Token" in Frage kommt. Passen auch die Ausdrücke für "Identifier" nicht zur aktuell eingelesenen Zeichenfolge, handelt es sich um eine unzulässige Zeichenfolge und der "Scanner" meldet den Fehler. Damit ist der Kompiliervorgang vorzeitig beendet und muss mit verbessertem Quelltext neu gestartet werden.

```
1  // Identifiers
2  IDENTIFIER: Letter LetterOrDigit*;
3
4  // Fragment rules
5  fragment ExponentPart
6      : [eE] [+-]? Digits
7      ;
8  fragment EscapeSequence
9      : '\\' [btnfr"'\\]
10     | '\\' ([0-3]? [0-7])? [0-7]
11     | '\\' 'u'+ HexDigit HexDigit HexDigit HexDigit
12     ;
13 fragment HexDigits
```

```
14      : HexDigit ((HexDigit | '_')* HexDigit)?
15      ;
16  fragment HexDigit
17      : [0-9a-fA-F]
18      ;
19  fragment Digits
20      : [0-9] ([0-9_]* [0-9])?
21      ;
22  fragment LetterOrDigit
23      : Letter
24      | [0-9]
25      ;
26  fragment Letter
27      : [a-zA-Z\$\_]
28      | ~[\u0000-\u007F\uD800-\uDBFF]
29      | [\uD800-\uDBFF] [\uDC00-\uDFFF]
30      ;
```

Quelltext 2.2: Reguläre Ausdrücke für Erkennung von "Identifiern" in "Java TX"-Quellcode

Die dargestellten "Tokens" und zugehörigen reguläre Ausdrücke stammen aus dem "Frontend" des "Java TX"-Compilers. Dort wird das Werkzeug "Antlr" verwendet, um den "Scanner" für die Analysephase aus regulären Ausdrücken zu generieren [46]. Diese Lösung vereinfacht die Entwicklung von "Java TX", da es den Entwicklern die manuelle Implementierung des "Scanners" in komplexen, endlichen Automaten erspart. Eine Regel besteht jeweils aus einem "Token" sowie einem oder mehreren regulären Ausdrücken, die durch den "|"-Operator voneinander getrennt sind. Genau wie der "Parser" priorisiert der "Scanner" die Ausdrücke einzelner "Token" anhand der Reihenfolge, in der sie auftauchen (vgl. 2.2.2 auf der nächsten Seite). Anstatt eines langen, komplexen regulären Ausdrucks kann dieser auch in einzelne Fragmente zerlegt werden. Diese Fragmente können dann zu komplexen Ausdrücken in einer "Scanner-Regel" zusammengesetzt werden (wie im obigen Ausschnitt zu sehen ist). Mithilfe der erkannten "Tokens" setzt der "Java TX"-Compiler die relevanten Elemente des Quelltextes zusammen und analysiert dessen Abfolge mittels "Parser" auf syntaktische Korrektheit.

2.2.2 Parser

Der "Parser" spielt in der Analysephase des "Frontends" im "Java TX"-Compiler eine zentrale Rolle. Er baut eine Datenstruktur auf, in der das zu verarbeitende Computerprogramm mit seinen relevanten, syntaktischen Elementen enthalten ist — den "Parsebaum" (siehe 2.2.2 auf Seite 17). Dabei prüft er die Eingabe auf syntaktische Korrektheit. Die Grundlage für die Arbeit des "Parsers" bildet eine sogenannte "kontextfreie Grammatik". Sie wurde im Rahmen von Forschungsarbeiten zu formalen Sprachen und Compilerbau entwickelt, um Syntaxregeln einer Programmiersprache zu definieren [39, Kapitel 3.2]. Es existieren verschiedene Notationsformen für kontextfreie Grammatiken. Die gebräuchlichste Form für die Verwendung im Bereich des Compilerbaus ist die "Erweiterte Backus-Naur-Form", die auch als Vorlage für die Definition der kontextfreien Grammatik des "Java TX-Parsers" verwendet wird [58, Kapitel 7.3.1 auf S. 235].

Grundsätzlich besteht eine kontextfreie Grammatik aus verschiedene Elementen. Zunächst gibt es die sogenannten "Terminale". Sie repräsentieren die vom "Scanner" erzeugten "Token" innerhalb der Grammatik des "Parsers" und bilden die kleinsten (beziehungsweise "feinsten") Elemente. Zusätzlich definiert eine kontextfreie Grammatik "Nicht-Terminale". Diese Elemente erlauben die Definition von Regeln, nach denen der Java "Parser" den Quelltext untersucht und können aus "Terminalen" und anderen "Nicht-Terminalen" zusammengesetzt sein. Sie repräsentieren alle erlaubten syntaktischen Elemente der Sprache "Java TX". Außerdem können bei der Definition der Grammatikregeln mit "Antlr" verschiedenen Operatoren verwendet werden. Ein optionales Element einer Regel kann beispielsweise mit dem "?"-Operator gekennzeichnet sein (siehe Ausschnitt in 2.4 auf Seite 15). Dieser Operator wird in der "Java TX"-Grammatik beispielsweise für die meisten Typangaben eingesetzt. Allgemein orientieren sich die Operatoren an regulären Ausdrücken. Der "*"-Operator steht für beliebig viele Wiederholungen eines grammatikalischen Elements. Der "+"-Operator erlaubt ein oder mehrere "Terminale" und "Nicht-Terminale" [46, Kapitel 5].

Die beschriebenen Elemente werden schließlich in Grammatikregeln miteinander kombiniert. Eine Grammatikregel besteht aus einem "Nicht-Terminal" auf ihrer linken Seite und einer Kombination aus "Nicht-Terminale", "Terminalen" und Operatoren auf ihrer rechten Seite (siehe 2.4 auf Seite 15) Während der syntaktischen Analyse des Quelltextes, bekommt der "Parser" einzelne "Token" vom "Scanner" übergeben. Die "Token" werden solange aneinandergereiht, bis sie mit der rechten Seite einer Grammatikregel übereinstimmen. In diesem Fall wendet der "Parser" die Regel an und ersetzt die Tokenfolge durch

das zugehörige "Nicht-Terminal" auf der linken Seite. Ist der eingegebene Programmtext syntaktisch korrekt, kann dieser Vorgang solange wiederholt werden, bis die oberste Regel der Grammatik angewendet werden kann. Sie repräsentiert die Wurzel des "Parsebaums", der die Syntax des Programms repräsentiert (vgl. 2.2.2 auf Seite 17), und signalisiert dem "Parser", dass die syntaktische Analyse erfolgreich abgeschlossen wurde.

Grammatik des Java-TX-Compilers

Die Grammatik des "Java TX"-Compilers bildet die Grundlage für den im Projekt eingesetzten "Antlr"-Parser. Das Tool "Antlr" erzeugt aus ihr einen "Top-Down-Parser". Die "Top-Down-Analyse" beginnt mit "Nicht-Terminalen", welche die grobe Struktur des Programmtextes darstellen und wird feiner, bis sie bei den "Terminalen" angekommen ist. "Bottom-up-Parser" verfahren in umgekehrter Weise. Der zugehörige "Parsetree" wird von der Wurzel aus konstruiert [39, Kapitel 5.3]. Bei "Antlr" handelt es sich um ein Werkzeug, dass im Compilerbau für die Generierung von "Scanner" und "Parser" verwendet werden kann. Die Syntaxregeln für eine vom Compiler akzeptierte Sprache können in einem übersichtlichen Format definiert werden. Aus den Regeln können anschließend Java-Klassen erstellt werden, welche für die syntaktische Analyse von Eingabetexten verwendet werden können. Dabei werden Klassen für sämtliche enthaltenen "Nicht-Terminale" erstellt. Der resultierende Parsebaum enthält Objekte dieser Klassen. Außerdem stellen sie Methoden bereit, um den Parsebaum zu untersuchen und zu modifizieren [46, Kapitel 1].

```
1  @SuppressWarnings("CheckReturnValue")
2          public static class InterfaceBodyDeclarationContext extends
               ↪ ParserRuleContext {
3                  public InterfaceBodyDeclarationContext(ParserRuleContext
                       ↪  parent, int invokingState) {
4                          super(parent, invokingState);
5                  }
6                  @Override public int getRuleIndex() { return
                       ↪ RULE_interfaceBodyDeclaration; }
7
8                  public InterfaceBodyDeclarationContext() { }
9                  public void copyFrom(InterfaceBodyDeclarationContext ctx
                       ↪ ) {
10                         super.copyFrom(ctx);
```

```
11                }
12        }
```

Quelltext 2.3: Beispiel für eine Klasse, die mithilfe des "Antlr"-Tools generiert wurde

Die aktuell verwendete Grammatik des "Java TX"-Compilers ist kompatibel mit Java Version 17. Sämtliche Sprachkonstrukte dieser Java Version werden vom "Java TX"-Parser erkannt und umgewandelt (vgl. 2.2.2 auf Seite 17). Jedoch weist die eingesetzte Grammatik eine Besonderheit auf. Da "Java TX" ohne Typangaben auskommt, akzeptiert der Parser der Sprache Quelltext ohne Typangaben. Lediglich in Sonderfällen müssen Typen angegeben werden (vgl. 2.6.1 auf Seite 43). Ansonsten sind Typangaben durch Entwickler optional. Die "Nicht-Terminale" für Typangaben sind in der Grammatik mit dem "?"-Operator, der für optionale Elemente steht, gekennzeichnet (siehe Regel "fieldDeclaration" in 2.4).

```
1  fieldDeclaration
2      : typeType? variableDeclarators ';'
3      ;
4
5  interfaceBodyDeclaration
6      : modifier* interfaceMemberDeclaration # interfacemember
7      | ';' # emptyinterface
8      ;
9
10 interfaceMemberDeclaration
11     : constDeclaration # interfaceconst
12     | interfaceMethodDeclaration # interfacemethod
13     | genericInterfaceMethodDeclaration # genericinterfacemethod
14     | interfaceDeclaration # subinterface
15     | annotationTypeDeclaration # interfaceannotationtype
16     | classDeclaration # interfaceclass
17     | enumDeclaration # interfaceenum
18     | recordDeclaration # interfacerecord // Java17
19     ;
20
21 constDeclaration
22     : typeType? constantDeclarator (',' constantDeclarator)* ';'
```

```
23      ;
```

Quelltext 2.4: Beispiel für Grammatikregeln für "Antlr" mit optionaler Typangabe und Labels

Zusätzlich zu den normalen Syntaxregeln bietet "Antlr" die Möglichkeit zur Verwendung von sogenannten "Labels". Sie dienen der Kennzeichnung von Alternativen für eine bestimmte Regel. Sind sämtliche Alternativen mit "Antlr"-Labels versehen (siehe Regel "interfaceMemberDeclaration" in 2.4 auf der vorherigen Seite), wird für jede Alternative eine separate Java-Klasse erzeugt. Diese Klasse erbt vom zugehörigen "Nicht-Terminal" der linken Seite. Dadurch können die Elemente des Parsebaums nach Abschluss der syntaktischen Analyse mittels "Pattern Matching" (vgl. 2.6.1 auf Seite 43) untersucht werden. Dies erleichtert die Umwandlung des Parsebaums in einen AST erheblich, da lange "if-else"-Konstrukte, die jede Alternative überprüfen, durch kompakte "switch"-Statements oder -Ausdrücke ersetzt werden können.

Rekursion & Operatorpräzedenz Beim Einsatz von "Top-Down-Parsern" entstehen verschiedene Schwierigkeiten, die mit der Natur kontextfreier Grammatiken zusammenhängen. Grammatiken sind grundsätzlich mehrdeutig [46, Kapitel 5, "Dealing with Precedence, Left Recursion, and Associativity"]. Mehrdeutigkeit liegt vor, wenn es für eine bestimmte Eingabe mehrere Ableitungsbäume gibt [39, Kapitel 5.2]. Diese Mehrdeutigkeit wird bei der Definition von Grammatikregeln für Ausdrücke besonders deutlich. Für Ausdrücke, wie arithmetische Operationen, Vergleiche, etc., werden oft links-rekursive Regeln erstellt. Das bedeutet, dass dasselbe "Nicht-Terminal" sowohl auf der linken Seite einer Regel, als auch auf deren rechten Seite steht. Ein Beispiel hierfür ist die Definition von Grammatikregeln für arithmetische Ausdrücke in Programmiersprachen. Die Vorstellung, dass ein Ausdruck aus zwei Ausdrücken besteht, die mit einem Operator verknüpft sind, lässt sich einfach in einer kontextfreien Grammatik implementieren (siehe unten stehendes Beispiel 2.5).

```
1  expr: expr '*' expr // match subexpressions joined with '*' operator
2  | expr '+' expr // match subexpressions joined with '+' operator
3  | INT // matches simple integer atom
4  ;
```

Quelltext 2.5: Beispiel für links-rekursive Grammatikregeln für arithmetische Ausdrücke aus Kapitel 5 in [46]

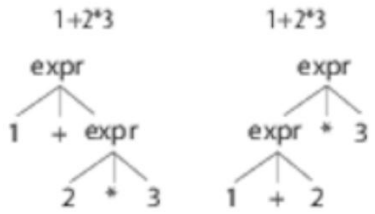


Abbildung 2.1: Zwei Möglichkeiten für einen "Parsetree" bei gleicher Eingabe
Quelle: Kapitel 5, Abschnitt "Dealing with Precedence, Left Recursion, and Associativity " in [46]

Diese Grammatikregel ist jedoch nicht eindeutig für "Top-Down-Parser". Die Präzedenz der Operatoren kann in kontextfreien Grammatiken standardmäßig nicht definiert werden. Bei der Verwendung beider Operatoren in einem Ausdruck ist nicht klar, welche Alternative zuerst angewendet werden soll. Es gibt also zwei Möglichkeiten für einen "Parsetree" (siehe Abbildung 2.1). Die linke Variante würde zuerst die Multiplikation ausführen und anschließend eins addieren. Das rechte Programm würde genau umgekehrt ablaufen. Der Parser muss mit diesem Problem umgehen können und die Definition von Operatorpräzedenzen ermöglichen. "Antlr" nutzt dafür eine ganz einfache Regel. Die Alternative, welche zuerst in der Grammatik definiert wurde, hat Vorrang. Im Falle unseres Beispiels hat die Multiplikation Vorrang vor der Addition. Diese Implementierung ist also korrekt [46, Kapitel 5].

Ableitungsbäume

Ableitungsbäume (engl. "Parse trees") repräsentieren die Syntax des analysierten Quelltextes. Der "Antlr"-Parser erstellt den "Parsetree" (oder "Parsebaum") aus Objekten der Klassen, welche die einzelnen Grammatikregeln repräsentieren (vgl. 2.3 auf Seite 14) [46, Kapitel 2, "Building Language Applications Using Parse Trees"]. Der "Parsetree" ist, im Falle des "Java TX"-Compilers, die erste Form der Zwischendarstellung eines Programms. Er stellt die syntaktischen Struktur des Programms detailliert dar und enthält Knoten für sämtliche "Nicht-Terminale" und "Terminale" beziehungsweise "Tokens", die im analysierten Quelltext vorkommen. In 2.6 auf der nächsten Seite ist der "Parsetree" des Statements `return lam1.apply(new Apply());` zu sehen. Sämtliche "Terminale" und "Nicht-Terminale", die "Parser" (und "Scanner") während der syntaktischen Analyse erkannt haben, sind zu enthalten. Der "Parsetree" für "applyLambda.jav" ist deutlich größer als die Quell-

datei selbst (siehe Anhang A auf Seite 71). Selbst für einen kurzen Quelltext nutzt der "Parser" bereits viele verschiedene "Nicht-Terminale" und "Terminale". Es ist fast selbsterklärend, dass eine solche Zwischendarstellung eines Computerprogramms nicht für die Analyse durch Menschen gedacht ist.

```
1  [...]
2                    blockStatement
3                      statement return
4                        expression
5                          expression
6                            primary
7                              identifier lam1
8                          .
9                          methodCall
10                           identifier apply
11                           (
12                           expressionList
13                             expression new
14                               creator
15                                 createdName
16                                   identifier Apply
17                                 classCreatorRest
18                                   arguments ()
19                           )
20                           ;
21 [...]
```

Quelltext 2.6: Auszug aus dem "Parsetree" einer "Java TX"-Quelldatei

Obwohl Computer (beziehungsweise Compiler) Ableitungsbäume problemlos verarbeiten können, wäre seine Nutzung als Zwischendarstellung, aus der JVM-"Bytecode" generiert wird, eher ineffizient [1]. Die Analyse und Weiterverarbeitung des "Parsetree"s wäre aufwendiger als die des eingesetzten AST (siehe Kapitel 2.4.1 auf Seite 22), da ein "Parsetree" viele Elemente enthält, die für den weiteren Kompilierungsvorgang irrelevant sind. Dazu zählen zum Beispiel "Terminale" wie Klammern ("(", ")") oder das Semikolon (";"). Sie dienen eher den Entwicklern, indem sie zum Beispiel die Lesbarkeit von Quelltext für Menschen steigern. Sie werden bei der Umwandlung von "Parsetree" zu AST entfernt (siehe

Abschnitt 2.4.1 auf Seite 22). Die Gesamtheit solcher Elemente wird auch als "Syntactic Sugar" bezeichnet.

2.3 "Syntactic Sugar"

Unter "Syntactic Sugar" versteht man Elemente einer Programmiersprache, die als alternative Schreibweise für bestehende Programmstrukturen dienen oder die allgemeine Lesbarkeit von Quelltext für die Entwickler verbessern sollen. Der Begriff wurde bereits 1964 bei der Untersuchung von Ausdrücken in Sprachen verwendet [42]. Ein Beispiel für die Anwendung von "Syntactic Sugar" in modernen Programmierpsrachen sind die Zuweisungsoperatoren "+=", "-=", "*=" und "/=". Sie werden zur Vereinfachung von Ausdrücken wie `x = x+3` verwendet. Semantisch bedeutet dieser Ausdruck, dass der aktuelle Wert der Variable x um 3 erhöht wird. Dieses Ergebnis wird in der Variable x gespeichert. Äquivalent dazu können Programmierer auch die genannten Operatoren verwenden. Der Ausdruck `x += 3` führt exakt zum selben Ergebnis. Syntaktisch wurde der Ausdruck `x = x+3` vereinfacht. Die Semantik bleibt unverändert. Der Compiler wird "Bytecode" für die arithmetische Operation (im Falle des Beispiels eine "Addition") und für die Zuweisung des Ergebnisses zur Variable x generieren.

Ähnlich wie die Abkürzung von Ausdrücken tragen auch Sonderzeichen zur Erleichterung des Verfassens von Quelltext bei. So werden beispielsweise geschweifte Klammern verwendet, um funktionale Blöcke in Programmierpsrachen voneinander abzugrenzen. Bei der Definition einer Funktion in "C" folgt die geschweifte Klammer auf die Liste von Parametern für die betreffende Methode. Nach der Klammer folgen beliebige (syntaktisch korrekte) Statements. Nach dem letzten Statement der Methode wird der "Block" beziehungsweise die Funktion durch die geschlossene, geschweifte Klammer abgeschlossen. Die geschweifte Klammer hat keinerlei Bedeutung für die Übersetzung eines Programmes in Maschinencode (beziehungsweise "Bytecode"). Gleiches gilt für runde und eckige Klammern. Sie sind integraler Bestandteil der Syntax einer Programmiersprache und tragen einen großen Teil zur Lesbarkeit eines Computerprogramms bei. Jedoch können sie nach der Syntaxprüfung vom Compiler verworfen werden, da sie den weiteren Kompilierungsprozess nicht beeinflussen.

```
1 int main(void){
2         int x = 0;
3         x *=2;
4         x +=3;
5         return x;
6 }
```

Quelltext 2.7: Beispiel für die Verwendung von "Syntactic Sugar" in "C"-Quelltext

"Syntactic Sugar" in Java

Auch Java verwendet syntaktische Elemente, deren Aufgaben die Unterstützung der Entwicklern und Steigerung der Lesbarkeit von Quelltext sind. Ein einfaches Beispiel hierfür ist die Möglichkeit, das "return"-Statement am Ende einer Methode nicht anzugeben. Der Compiler erkennt fügt fehlende "return"-Statements automatisch in den AST ein, ohne dass eine Korrektur des Quelltextes durch die Entwickler vorgenommen werden muss. Im "Java TX"-Compiler übernimmt diese Aufgabe eine Subkomponente des "SyntaxTreeGenerators" (vgl. Abschnitt 2.4.1 auf Seite 22).

Weiterhin gibt es, neben den im vorherigen Abschnitt beschriebenen Elemente, in Java auch eine sogenannte "foreach"-Schleife. Klassische "for"-Schleifen werden, vor allem in objektorientierten Sprachen, oft für das Iterieren über eine Sammlung von Elemente verwendet (beispielsweise Elemente eines Arrays oder einer Liste). Für diesen Anwendungsfall erlaubt die abgewandelte "foreach"-Schleif eine komfortable Syntax (siehe folgender Quelltext 2.8).

```
1 void printNamesWithSugar(){
2         ArrayList<String> names = Arrays.asList(``Jason'', ``Pierre'',
              ↪ ``Rick'', ``Michael'');
3         for(String name : names){
4                 System.out.println(name);
5       }
6 }
```

Quelltext 2.8: Beispiel für eine "foreach"-Schleife: Alle Elemente der Liste "names" werden ausgegeben

Die Vereinfachung für Programmierer besteht darin, dass Zählvariablen für das Durchlaufen einer Sammlung von Elementen automatisch bereitgestellt werden. Außerdem steht das aktuelle Elemente in jedem Schleifendurchgang unter einem definierten Namen (hier "name") zur Verfügung. In einer klassischen "for"-Schleife ist es die Pflicht der Programmierer, Variablen zu deklarieren, welche die Schleifendurchläue zählen und auf die Elemente der Sammlung zu zugreifen (siehe folgender Quelltext 2.9). Das Ergebnis der Übersetzung dieser beiden Schleifen in Java "Bytecode" ist nahezu identisch (vgl. Bytecode in Anhang B auf Seite 75). Sie unterscheiden sich lediglich in ihrer Syntax.

```
1 void printNames(){
2         ArrayList<String> names = Arrays.asList(``Jason'', ``Pierre'',
              ↪ ``Rick'', ``Michael'');
3         for(int i = 0; i < names.size(); i++){
4                 System.out.println(names.get(i));
5         }
6 }
```

Quelltext 2.9: "foreach"-Loop als klassische "for"-Schleife

Ein weiteres Beispiel für "Syntactic Sugar" sind die in Java Version 16 eingeführten "Records". Sie sollen die Deklaration von Klassen, deren Hauptaufgabe die Speicherung von Daten ist, vereinfachen [32]. Sie werden im Rahmen dieser Arbeit in "Java TX" integriert, so dass der Compiler die Syntax erkennt und entsprechende Knoten in den AST einfügen kann (siehe Kapitel 2.6.1 auf Seite 38 und Kapitel 3.2.1 auf Seite 50).

2.4 Zwischendarstellung eines Computerprogramms

Wie bereits in Kapitel 2.1 auf Seite 5 beschrieben, stellt die Zwischendarstellung eines Computerprogramms die Grundlage für die vom Backend des Compilers durchgeführte Synthesephase dar. Der im obigen Abschnitt beschriebene "Parsetree" (engl. "Parse tree") dient als Grundlage für die Generierung der Zwischendarstellung eines "Java TX"-Programms. Er wird von einer Komponente des Compiler-Frontends, dem "SyntaxTreeGenerator", analysiert und umgewandelt. Während dieser Analyse werden bereits erste Typchecks durchgeführt und das Programm "wird für die Typinferenz vorbereitet" (siehe Abschnitt 2.4.1 auf der nächsten Seite). Es findet also ein erster Teil der semantischen Analyse statt. Aus

den Ergebnissen der Analyse erstellt der "SyntaxTreeGenerator" dann einen AST für das Programm (mehr dazu in Abschnitt 2.4.1).

Der Grund für die weitere Verarbeitung des "Parsetree"s ist, dass dieser zahlreiche Details enthält, die keine Informationen über den tatsächlichen Programmablauf enthalten und keinen Einfluss auf den generierten "Bytecode" haben [42]. Andererseits fehlen die Ergebnisse der semantischen Analyse, der zweiten Phase des Compiler Frontends (vgl. 2.1 auf Seite 5). Dabei geht es hauptsächlich um die Prüfung der im Programm enthaltenen Typangaben und die Berechnung eventuell notwendiger Typkonversionen. Außerdem setzt der "Java TX"-Compiler während der semantischen Analyse den Typinferenzalgorithmus ein, um fehlende Typangaben mithilfe von Bedingungen ("Constraints") zu berechnen (siehe 2.5.2 auf Seite 27). Der "Parsetree" muss also transformiert werden, um eine vollständige Zwischendarstellung des Computerprogramms zu erzeugen (siehe folgende beiden Abschnitte). Erst dann kann das "Backend" mit der Generierung des "Bytecodes" beginnen.

2.4.1 Abstrakte Syntaxbäume

Der AST enthält sämtliche relevanten, syntaktischen Informationen über ein Computerprogramm. Er dient dem Compiler als Zwischendarstellung des zu übersetzenden Quelltextes. Der Baum wird während der syntaktischen Analyse des Quelltextes aufgebaut. Die Baumstruktur kann gut von Computern verarbeitet werden. Bäume sind wichtige Datenstruktuen in der Informatik [35, Kapitel 3.6]. Jeder Knoten des Baumes steht für ein syntaktisches Element des Quelltextes. An der Wurzel steht ein Knoten, der das komplette Programm, beziehungsweise eine komplette Quelldatei, repräsentiert. Die verschiedenen Teilbäume stehen dann für die einzelnen Bestandteile, die im Quelltext definiert wurden (z.B. Klassen, Methoden, Felder, Ausdrücke, etc.).

Transformation von "Parse-" zu "Syntaxbäumen"

Wie bereits in Abschnitt 2.3 auf Seite 19 beschrieben, wird der "Syntactic Sugar" eines Programms nicht im AST gespeichert. Er ist für die weitere Übersetzung des Programms irrelevant. Deshalb transformiert die Komponente "Syntaxtreegenerator" des "Java TX"-Compilers den detaillierten "Parsetree" des "Parsers" zu einem AST. Dadurch ensteht der erste Teil einer kompakten Zwischendarstellung — bestehend aus AST und Symboltabelle (siehe folgender Abschnitt 2.4.2 auf Seite 24) [39].

Der "Syntaxtreegenerator" nutzt den "Parsetree" als Eingabe, filtert irrelevante Informationen heraus und erstellt für jedes syntaktische Konstrukt einen Knoten im AST. Außerdem ergänzt er syntaktische Elemente, die nicht oder nicht vollständig im "Parsetree" des "Parsers" enthalten sind. Ein Beispiel hierfür sind die "TypePlaceholder", die während der Typinferenz benötigt werden (siehe Kapitel 2.5.2 auf Seite 27). Das Ergebnis ist ein vollständiger, AST aller Quelldateien des aktuellen Kompilierungsvorgangs.

```
1  TPH m(){
2    TPH lam1;
3    lam1 = (TPH x) -> {
4      return x;
5    };
6    return lam1.apply Signature: [TPH, TPH](new Apply());
7  }
```

Quelltext 2.10: Auszug aus dem AST der Quelldatei in Anhang A auf Seite 71

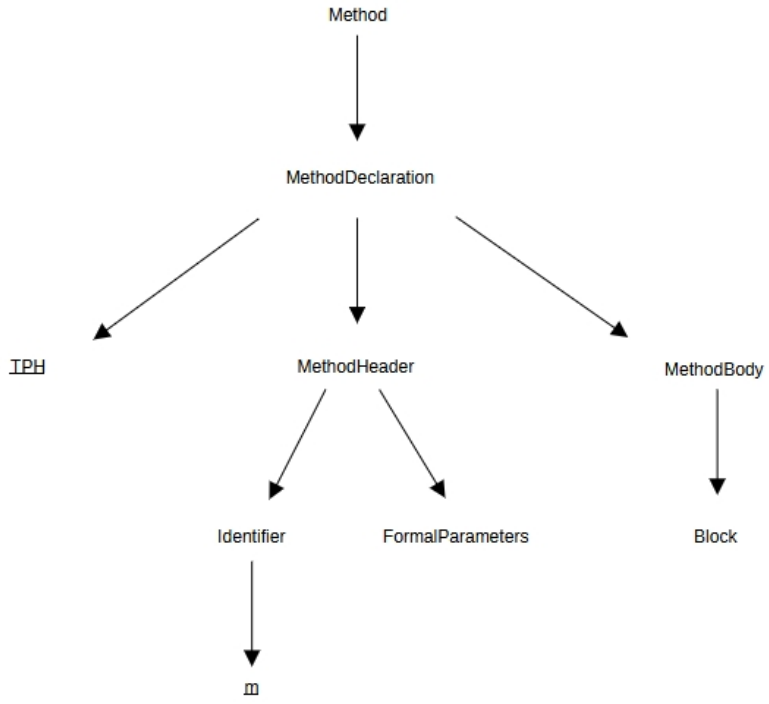


Abbildung 2.2: Schematische Darstellung des AST für Zeile 1 des Quelltextes 2.10

Der "Syntaxtreegenerator" wurde bereits, mit der Einführung der neuen Grammatik für

Java Version 17, aktualisiert. Im Rahmen dieser Arbeit wird seine korrekte Funktionsweise durch die Entwicklung von Unit-Tests überprüft (siehe Kapitel 3.3 auf Seite 65).

2.4.2 Symboltabelle

Die Symboltabelle enthält sämtliche Symbole, die im Quelltext eines Computerprogramms definiert wurde und kommt im klassischen Compilerbau zur Anwendung [39, Kapitel 6.6] [55, Kapitel 5.5]. Im Falle von Java (und "Java TX") zählen dazu Methoden, Variablen und Konstanten. Jedem Eintrag sind Attribute zugeordnet, die der Compiler zur korrekten Übersetzung des Quellcodes benötigt. Die Einträge werden während der Analyse des Programms durch das Frontend erstellt. Im Falle von "Java TX" geschieht dies während der Typinferenz (vgl. Kapitel 2.5.2 auf Seite 27). Je nach Programmiersprache können auch noch die Speicheradressen in der Symboltabelle enthalten sein. Der Eintrag einer Methode unterscheidet sich insofern von anderen Symbolen, als dass in der Symboltabelle sowohl der Typ ihres Rückgabewertes, als auch die Typen sämtlicher Parameter gespeichert sind [39, Kapite 6]. "Java TX" führt sogar neue Interfaces ein, um echte Funktionstypen für Lambda-Ausdrücke zu ermöglichen (vgl. Kapitel 2.5.3 auf Seite 31).

Die Symboltabelle wird im Frontend des "Java TX"-Compilers erstellt. Da "Java TX" das Java-Typsystem erweitert, gibt es keine klassische Symboltabelle. Sämtliche Typen werden vom implementierten Typinferenzalgorithmus berechnet. Deshalb ist die Symboltabelle als sogenanntes "ResultSet" implementiert. Ein "ResultSet" enthält mehrere "ResultPairs", die Ergebnisse der Typunifikation repräsentieren (siehe 2.5.2 auf Seite 29). Bevor der Typinferenzalgorithmus die Typen der Bezeichner eines Programms berechnet hat, werden für nicht deklarierte Typen sogenannte "TypePlaceholder" eingesetzt. Diese sind auch als Knoten im AST enthalten. Bei der Übersetzung des AST im "Backend" wird das "ResultSet" der Typinferenz genutzt, um Typen von Symbolen aufzulösen, ihren korrekten Einsatz zu überprüfen und den Quellcode, der das Symbol verwendet, korrekt in "Bytecode" zu übersetzen [57] [49].

2.5 Java TX

"Java TX" erweitert die Programmiersprache Java um globale Typinferenz, echte Funktionstypen, allgemeine sowie Durchschnittstypen (siehe Abschnitte 2.5.2 auf Seite 27). Sie

verfolgt das Ziel die Programmierung in Java einfacher und effizienter zu gestalten [37]. Der implementierte Typinferenzalgorithmus nutzt sogenannte "Type placeholder", um auf Basis der im Quellcode genutzten Datentypen für Variablen und Methoden Beschränkungen (engl. "constraints") zu generieren. Daraus kann dann jeder verwendeten Variable und Methode ein möglichst allgemeiner Typ zugeordnet werden.

Definition 2 *Typinferenz: Unter Typinferenz versteht man die automatische Berechnung von Typen für Ausdrücke in Programmiersprachen oder mathematischen Typsystemen [59]. Im Rahmen von Programmiersprachen erlaubt die Typinferenz das Verzichten auf Typangaben im Quelltext [49]. Es wird zwischen lokaler und globaler Typinferenz unterschieden. Lokale Typinferenz bezieht sich auf einzelne Codebausteine (Methoden). Globale Typinferenz umfasst den gesamten Quelltext eines Programms.*

```
1  import java.util.Vector;
2  import java.util.Stack;
3
4  class Put {
5
6      putElement(ele, v) {
7              v.addElement(ele);
8      }
9
10     putElement(ele, s) {
11             s.push(ele);
12     }
13
14
15     main(ele, x) {
16         putElement(ele, x);
17     }
18
19 }
```

Quelltext 2.11: Einfaches Beispiel für "Java TX" Code

Das Konzept der Typinferenz in Java ist nicht vollkommen neu. Im Kontext lokaler Variablen und bei der Deklaration von generischen Klassen ("Generics") kommen bereits Algorithmen

für die Berechnung von Datentypen zum Einsatz [53] [56]. Der im "Java TX"-Compiler integrierte Typinferenzalgorithmus ermöglicht jedoch die Programmierung in Java ohne die Angabe von Datentypen für Variablen oder Methoden. Sämtliche Datentypen der in einem "Java TX"-Programm enthaltenen Variablen und Methoden werden vom Typinferenzalgorihmus auf Basis der zu Beginn importierten Typen berechnet. Detaillierte Erläuterungen zur Funktionsweise des Algorithmus folgen in Abschnitt 2.5.2 auf der nächsten Seite.

Definition 3 *Funktionstypen: Der Typ einer Funktion hängt, im Bereich der Programmiersprachen, von den Typen des Definitions- und Wertebereichs einer Funktion ab. Er wird üblicherweise mit $A \to B$ angegeben, wobei A den Typ des Definitionsbereichs und B den Typ der resultierenden Werte einer Funktion beschreibt. Normalerweise beschreibt ein Funktionstyp den Typ einer Variable oder eines Parameters, der bzw. dem das Resultat einer Funktion zugeordnet wurde. Bei Funktionen höheren Grades kann er jedoch auch den Typ einer Funktion angeben, die als Parameter oder Rückgabewert dient.*

Neben der globalen Typinferenz unterstützt "Java TX" außerdem echte Funktionstypen. Mit der Einführung von "Lambda Expressions" in Java Version 8 wurden sogenannte "funktionale Interfaces" eingeführt, welche den Typ eines solchen Ausdruckes repräsentieren. Diese Art von Typdeklaration für "Lambda Expressions" hat verschiedene Nachteile und führt zu schwer nachvollziehbaren Ergebnissen bei der Berechnung von Typen durch den integrierten Typinferenzalgorithmus [49]. Deshalb wurden echte Funktionstypen, wie sie in der Programmiersprache "Scala" zu finden sind, in "Java TX" integriert [48]. Weitere Ausführungen zu echten Funktionstypen folgen in Abschnitt 2.5.3 auf Seite 31

2.5.1 Stand der Entwicklung

Zurzeit basiert "Java TX" auf Version 8 der Java Spezifikation [54]. Der "Java TX"-Compiler kann deren Syntax vollständig verarbeiten. Verantwortlich für die syntaktische Analyse des "Java TX"-Quellcodes ist ein Parser, der mithilfe des Tools "Antlr" erstellt wird. Der Parser wird aus einer speziellen Datei generiert, die eine kontextfreie Grammatik für die Programmiersprache Java enthält. Die zugehörige kontextfreie Grammatik wurde für die Verwendung mit "Java TX" modifiziert. Das heißt, alle Typangaben für Variablen, Methoden, etc. sind optional.

Definition 4 *Exception: Eine Exception (dt. "Ausnahme") wird in Computerprogrammen für den Austausch von Informationen zu Programmzuständen verwendet. Dabei handelt es sich meist um Fehlerzustände, welche Bedingungen für die Semantik einer Programmiersprache verletzen. Können Exceptions bewusst generiert und im Programm abgefangen beziehungsweise behandelt werden, spricht man von "Checked Exceptions". In bestimmten Fällen führen Exceptions zum vorzeitigen Abbruch des Programms [54, Kapitel 11].*

Trotz der syntaktischen Verarbeitung bestimmter Funktionen der Sprache Java sind diese semantisch nicht in "Java TX" implementiert und der Compiler generiert eine Exception, wenn nicht unterstützte Ausdrücke im Output des Parsers (Parsebaum, vgl. Abschnitt 2.2.2 auf Seite 17) enthalten sind. Diese "NotImplementedException" wird vom "Java TX"-Compiler erstellt, wenn ein Entwickler Funktionen nutzen möchte, die vom Compiler semantisch noch nicht unterstützt werden.

2.5.2 Typinferenz

"Java TX" nutzt einen eigens am Standort Horb der DHBW Stuttgart entwickelten Algorithmus zur Typinferenz. Er arbeitet mit dem AST, der aus dem Parsebaum des Quelltextes generiert wird (vgl. 2.3 auf Seite 34).

Definition 5 *Gebundene Typparameter: Gebundene Typparameter (engl. "Bounded Type Parameters") erlauben die Eingrenzung von Typen für generische Typparameter. Durch die Schlüsselworte "extends" und "super" wird ein Supertyp beziehungsweise Subtyp für den deklarierten Typparameter vorgegeben. Dies erlaubt bspw. den Einsatz bekannter Methoden des Supertyps auf die Instanz der generische Typvariable, obwohl der exakte Typ zum Zeitpunkt der Deklaration unbekannt ist [2].*

Definition 6 *Typen in "Java TX"-Programmen: Die folgenden Formalen Definitionen beschreiben Typen von "Java TX"-Elementen, die mit einem Typ beziehungsweise Bezeichner versehen werden können.*

- $v : \theta$*: Typ einer lokalen Variable* v

- $cl < \overline{T < T'} > .f : \theta$*: Typ des Feldes* f *der Klasse* cl *mit generischem Typparameter* T*, die durch Typ* T' *gebunden ist (Operator <)*

- $cl < \overline{T < T'} > .m :< \overline{R < R'} > (\theta_1, \ldots, \theta_n) \rightarrow \theta$*: Typ der Methode* m *aus Klasse* cl*. Für die Methode ist ein generischer Typparameter* R *definiert, der durch* R' *gebunden ist. Für die Klasse* cl *existiert zusätzlich noch der Typparameter* T *mit der Beziehung* $T < T'$

- $cl < \overline{T < T'} > .m :< \overline{R_1 < R'_1} > (\theta_{1,1}, \ldots, \theta_{1,n}) \rightarrow \theta_1 \& \ldots \& < \overline{R_m < R'_m} > (\theta_{m,1}, \ldots, \theta_{m,n}) \rightarrow \theta_m$*: Durchschnittstyp einer überladenen Methode* m *in Klasse* cl

Definition 7 *Überladung (engl. "Overloading"): Unter Überladung versteht man die mehrfache Deklaration von Methoden mit unterschiedlichen Signaturen. Diese beschreiben die Typen des Rückgabewerts sowie der erwarteten Parameter einer Methode [54, Kapitel 8.4.9].*

Der Algorithmus kann Typen im globalen Geltungsbereich eines "Java TX"-Programms durch die Definition von Bedingungen berechnen. Diese Bedingungen werden verwendet, um sogenannte "Principal Types" zu inferieren. Die Berechnung der Typen eines "Java TX"-Programms erfolgt in zwei Schritten.

Durchlaufen des Syntaxbaumes

Der Algorithmus durchläuft zuerst den AST des Programms und sammelt Informationen über enthaltene Typen beziehungsweise "TypePlaceholder". Diese Informationen nutzt der Algorithmus, um Beziehungen zwischen den unbekannten Typen (den "TypePlaceholdern") zu erstellen und Bedingungen für deren Berechnungen aufzustellen (sog. "Constraints"). Wird beispielsweise eine Variable x mit einem primitiven Integer initialisiert, kann der Typinferenzalgorithmus bei einer Zuweisung von x zu einer weiteren Variable ohne Typangabe y die Bedingung aufstellen, dass y ein Subtyp des Typs von x sein muss. In diesem Fall wäre die Bedingung also TPH Y $\lessdot$ $java.lang.Integer$. Wobei $\lessdot$ angibt, dass die beiden Typangaben vereint (engl. "unified") werden sollen [57].

Definition 8 *Subtyp: Ein Subtyp beschreibt einen Datentyp in der theoretischen Informatik. Ein Subtyp ist mit einem weiteren Datentyp, dem Supertyp, verknüpft. Die beiden Typen haben gemeinsame Elemente, wie Felder oder Methoden, die sowohl auf Objekte*

des Supertyps als auch auf Objekte des Subtyps angewendet werden können. Das bedeutet, dass Objekte des Supertyps im Kontext des Subtyps verwendet werden können [47, Kapitel 19.3] [54, Kapitel 4.10]. Die Beziehung zwischen Supertyp und Subtyp wird mit $\leq^*$ angegeben und ist in [57, Definition 1] formal definiert.

Nach diesem Prinzip wird die Menge $\{\overline{ty \lessdot ty'}\}$ generiert, die Bedingungen für zu berechnende Typen enthält. Diese Bedingungen beschreiben einzelne Typen gemäß Java Spezifikation [54, Kapitel 18]. Der Ausdruck `Arrays.asList(1, 2.0)` führt beispielsweise zu den Bedingungen

- $< 1 \lessdot \alpha >$
- $< 2.0 \lessdot \alpha >$

Durch Reduktion erhält man die Beschränkungen $Integer \lessdot \alpha$ und $Double \lessdot \alpha$. Diese sagen aus, dass der gesuchte Typ der Array-Elemente Supertyp für den Typ "Integer" und den Typ "Double" sein muss. Dieses Verfahren wird als Typunifikation für alle Bedinungen der Menge angewendet.

Typunifikation

Während der Unifikation berechnet der globale Typinferenzalgorithmus Substitutionsregeln für "TypePlaceholder", die vom "Syntaxtreegenerator" in den AST eingefügt worden sind (vgl. 2.4.1 auf Seite 22). Die Substitution von "TypePlaceholdern" wird iterativ für eine Untermenge von Typbedingungen ("Constraints") generiert. Die verbleibenden "Constraints" sind im Ergebnis der Berechnung enthalten [49, Abschnitt 3.1]. Der Prozess der Typunifikation soll an folgendem "Java TX"-Codebeispiel verdeutlicht werden:

```
1 class Fac {
2   N getFac (O n) {
3     P res = 1;
4     R i = 1;
5     while ((i::R) <= (n::O))::T {
6       (res::P)=(( res::P)*(i::R))::U ;
7       (i::R)++;
8     }
9     return(res::P);
```

```
10    }
11 }
```

Quelltext 2.12: "Java TX"-Quelltext für die Berechnung der Fakultät einer Zahl
Quelle: Beispiel 3.5 in [49]

Das Codebeispiel enthält bereits die "TypePlaceholder", die während der Umwandlung des "Parsetrees" in den AST eingefügt wurden. Daraus generiert der Typinferenzalgorithmus folgende Bedingungen [49, Beispiel 3.5]:

$$\begin{gathered}\{(P \doteq N),\\ (U \lessdot P),\\ (O \lessdot java.lang.Number),\\ (R \lessdot java.lang.Number),\\ (java.lang.Boolean \doteq T),\\ (java.lang.Integer \doteq U),\\ (R \lessdot java.lang.Integer),\\ (P \lessdot java.lang.Integer)\}\end{gathered}$$

Die anschließende Berechnung der Substitutionsregeln mittels Typunifikation führen zu folgendem Ergebnis:

$$\begin{gathered}\{\varnothing, [(U \to java.lang.Integer),\\ (P \to java.lang.Integer),\\ (R \to java.lang.Integer),\\ (O \to java.lang.Integer),\\ (N \to java.lang.Integer),\\ (T \to java.lang.Boolean)]\}\end{gathered}$$

Das Ergebnis enthält zwei Elemente. Die leere Menge gibt an, dass keine Bedingungen verbleiben. Sämtliche Typen konnten inferiert werden. Die Substitutionsregeln bilden den "Unifier" σ, das zweite Element des Ergebnisses. Wendet man diese Regeln an, erhält man den Quelltext 2.13 auf der nächsten Seite. In diesem Beispiel ist der berechnete "Unifier" der einzig mögliche. Grundsätzlich erlaubt "Java TX" durch den erweiterten

"Overloading"-Mechanismus auch mehrere Typangaben für dieselbe Deklaration (bspw. einer Funktion) [49, Kapitel 4]. Verbleiben, neben dem "Unifier", Bedingungen im Ergebnis der Typunifikation, würden generische Typvariablen für die "TypePlaceholder" eingesetzt werden. Der Gültigkeitsbereich dieser Variablen richtet sich nach der Position des "TypePlacholders" im Quelltext (innerhalb einer Methode oder einer Klasse) [49, Kapitel 3.1].

```
1  class Fac {
2    Integer getFac (Integer n) {
3      Integer res = 1;
4      Integer i = 1;
5      while ((i::Integer) <= (n::Integer))::Boolean {
6        (res::Integer)=((res::Integer)*(i::Integer))::Integer ;
7        (i::Integer)++;
8      }
9      return(res::Integer);
10   }
11 }
```

Quelltext 2.13: Ergebnis nach Anwendung der berechneten Substitutionsregeln für "TypePlacholder"
Quelle: Beispiel 3.5 in [49]

2.5.3 Echte Funktionstypen

Neben den Funktionen zur Inferenz von Typen in "Java TX"-Programmen unterstützt die Javaerweiterung auch sogenannte echte Funktionstypen für Lambda-Ausdrücke. Sie orientieren sich an der Implementierung der funktionalen Programmiersprache "Scala", erhalten aber die Vorteile der funktionalen Interfaces, die mit Lambda-Ausdrücken in Java Version 8 eingeführt wurden [48] [54, Kapitel 9.8]. Neben dem Einsatz funktionaler Interfaces, anstatt echter Funktionstypen, in Java, gaben zwei weitere Beschränkungen der Sprache Anlass zur Implementierung echter Funktionstypen. Erstens, die Subtyp-Beziehung (vgl. 8 auf Seite 28) ist für funktionale Interfaces nicht korrekt [49, Beispiel 6.1]. Außerdem können Ausdrücke wie `(x -> h(x)).apply(arg);` in Standard-Java nicht verwendet werden, da der Typ des Lambda-Ausdrucks nicht explizit deklariert wurde und daher nicht bekannt ist, ob die Methode `apply` existiert. Es gibt zwar Lösungsansätze für diese Probleme,

die Entwickler von "Java TX" entschieden sich aber stattdessen für die Einführung echter Funktionstypen [49, Kapitel 6].

Die Funktionstypen sind als spezielle Interfaces implementiert (vgl. 2.14). Sie werden für Lambda-Ausdrücke explizit definiert und unterstehen folgenden Bedingungen [49, Kapitel 6]:

- $FunN\$\$ < T'_1 \dots T'_N, T_0 > \leq^* FunN\$\$ < T_1 \dots T_N, T'_0 >$ sofern $T_i \leq^* T'_i$
- Es dürfen keine Wildcards in $FunN\$\$$-Typen vorkommen

```
interface FunN\$\$ <-T1 , ... , -TN, +R> {
  R apply (T1 arg1 , ... , TN argN);
}
interface FunVoidN$$ <-T1, ..., -TN> {
  void apply (T1 arg1, ..., TN argN);
}
```

Quelltext 2.14: Implementierung der Interfaces zur Realisierung echter Funktionstypen. Quelle: Kapitel 6 in [49]

Die Funktionstypen werden vom Typinferenzalgorithmus bestimmt und als Teil des "ResultSets", für die Erzeugung des Bytecodes im "Backend", gespeichert. Die Generierung des Typs erfolgt mithilfe der Signatur des Lambda-Ausdrucks. N gibt die Anzahl der Parameter der Funktion an. Für jeden Parameter wird zusätzlich ein Typ inferiert, der, zusammen mit dem Typ des Rückgabewertes, als generischer Typparameter an das Interface übergeben wird [48, Kapitel 6]. Folgendes Beispiel soll das Vorgehen des Typinferenzalgorithmus verdeutlichen:

```
class MatrixOP extends Vector<Vector<Integer>> {
  mul = (m1, m2) -> {
    var ret = new MatrixOP();
    var i = 0;
    while ( i < m1.size()) {
      var v1 = m1.elementAt(i);
      var v2 = new Vector<Integer>();
      var j = 0;
      while ( j < v1.size()) {
```

```
10          var erg = 0;
11          var k = 0;
12          while ( k < v1.size()) {
13            erg = erg
14            + v1 .elementAt(k)
15            * m2 .elementAt(k)
16            .elementAt(j);
17            k ++;
18          }
19          v2.addElement(erg);
20          j ++;
21        }
22        ret.addElement(v2);
23        i ++;
24      }
25      return ret;
26    }
27  }
```

Quelltext 2.15: "Java TX"-Klasse mit Lambda-Ausdruck
Quelle: Kapitel 7 in [49]

Aus dem Quelltext 2.15 auf der vorherigen Seite inferiert "Java TX" den Funktionstyp mul : $Fun2\$\$ < Vector <_? Vector <_? Integer >>, Vector <_? Vector <_? Integer >>, Matrix >$. Dieser weist zwar noch eine gewisse Komplextität auf, ist jedoch deutlich weniger komplex als die Typen, die Java mithilfe der Standard-Interfaces generieren würde [49, Kapitel 6].

2.5.4 Kompilierung von Quelldateien

Nach detaillierter Beschreibung des Frontends des "Java TX"-Compilers sollen die beschriebenen Komponenten in den Kontext des vollständigen Kompilierungsprozesses eingeordnet werden. Abbildung 2.3 auf der nächsten Seite stellt diesen Prozess schematisch dar. Um die Übersichtlichkeit zu wahren, wurden nur die Hauptkomponenten in die Abbildung eingefügt. Sämtliche Subkomponenten und -klassen können der Übersicht über die Projektstruktur in Anhang C auf Seite 77 entnommen werden.

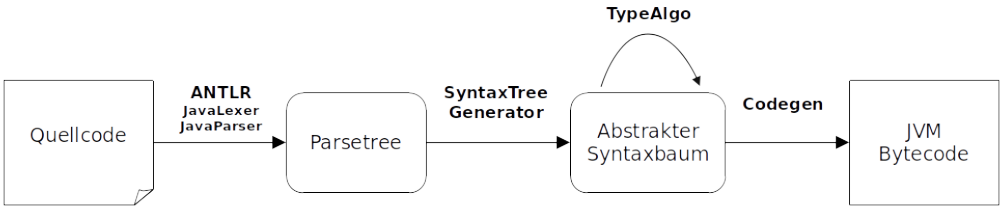


Abbildung 2.3: Darstellung des Kompiliervorganges eines "Java TX"-Programmes und der verantwortlichen Komponenten des Compilers

Der Kompilierungsvorgang beginnt mit dem Einlesen von Quellcode, der auf mehrere Quelltextdateien verteilt sein kann. Der von "Antlr" generierte Parser überprüft den Quellcode auf syntaktische Korrektheit (vgl. Kapitel 2.2 auf Seite 9). Sofern der Quellcode keine Syntaxfehler aufweist, beginnt der "SyntaxTreeGenerator" mit der Umwandlung der "Parsetrees" in ASTs (einer pro Quelltextdatei). Die Besonderheit des "SyntaxTreeGenerators" ist, dass er nicht ausschließlich die Umformung durchführt, um nicht benötigte Informationen des "Parsetrees" zu entfernen. Zusätzlich beginnt der "SyntaxTreeGenerator" bereits mit einem Teil der semantischen Analyse des Programms. Er führt Typprüfungen durch, berechnet Typumwandlungen und bereitet den AST durch das Einfügen von "TypePlaceholdern" für den zweiten Teil der semantischen Analyse vor 2.4.1 auf Seite 22. Dieser Teil ist die große Besonderheit des "Java TX"-Compilers. Der Typinferenzalgorithmus berechnet die fehlenden Typen des Programms und speichert sie als "ResultSet" (vgl. Kapitel 2.5.2 auf Seite 27). Sobald der Algorithmus seine Berechnungen abgeschlossen hat, beginnt die Komponente "Codegen" mit der Generierung des JVM Bytecodes. Sie nutzt dafür die Informationen des AST und kombiniert diese mit den Ergebnissen des Typinferenzalgorithmus. So können "TypePlaceholder" durch berechnete Typen ersetzt werden. Der generierte Bytecode ist konform zur Spezifikation der JVM und kann in dieser Umgebung ausgeführt werden (siehe Anhang D auf Seite 82) [43].

2.6 Entwicklung der Sprache Java seit Version 8

Programmiersprachen sind sehr dynamisch und entwickeln sich aufgrund neuer Erkenntnisse und Anforderungen ständig weiter [39, Kapitel 1]. Dementsprechend unterliegen auch die zugehörigen Compiler diesen Entwicklungen. Gerade Open-Source-Projekte mit vielen

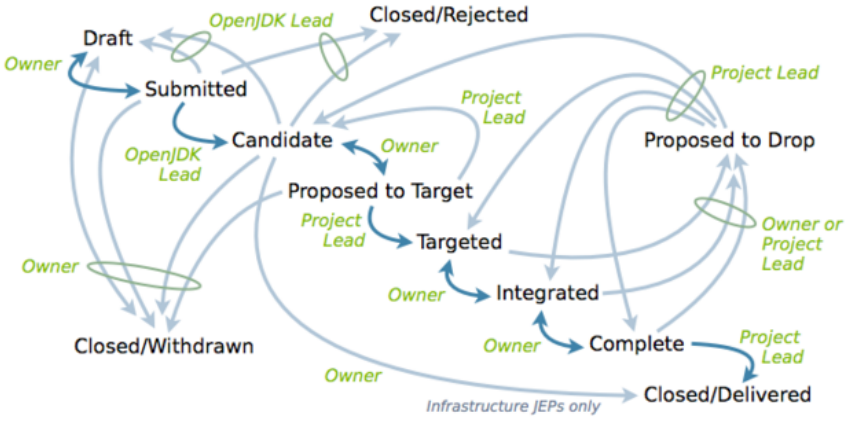


Abbildung 2.4: Darstellung des Lebenszyklus eines JEPs
Quelle: `https://cr.openjdk.org/~mr/jep/jep-2.0-02.html`

mitwirkenden Entwicklern verändern sich in schnellem Tempo. Für Java erscheint beispielsweise alle zwei Jahre ein neuer "Release" mit Long Term Support (LTS) [4]. In diesem Zeitraum erscheinen Updates für die LTS-Versionen, um Bugs zu beheben oder Sicherheitslücken zu schließen. Außerdem werden sogenannte "Feature Releases" veröffentlicht, um neue Funktionen in die Programmiersprache aufzunehmen [3]. Die Diskussion über neue Funktionen für das Open Source Java Development Kit (JDK) findet mittels JEPs statt. Sie erlauben jeder interessierten Person, Ideen für die Erweiterung und/oder Verbesserung der Programmiersprache Java vorzuschlagen. Natürlich gehören zu den Vorschlägen auch konkrete Ideen für die Implementierung und Pläne für die Integration in einen geplanten Java "Release" [50]. Folgende Grafik 2.4 stellt den Ablauf der Bearbeitung eines JEPs dar. Je nach Dimension der vorgeschlagenen Änderungen kann dieser Prozess mehrere Monate dauern. Zum Beispiel, wenn es sich um eine neue Funktion für die Programmiersprache handelt, die von Grund auf implementiert werden muss.

Es existieren rund 300 JEPs, die neue Funktionen für Java nach JDK8 vorschlagen. Einige davon wurden durch die führenden Java Entwickler ("Java Lead") abgelehnt oder erwiesen sich während der Bearbeitung als nicht umsetzbar. Viele wurden aber in verschiedenen "Releases" in die Programmiersprache aufgenommen und im JDK (also auch im Compiler) implementiert. Sie reichen von der Erweiterung des Ein-/Ausgabe-Systems, über verbesserte

Fehlerbehandlung, bis hin zur Einführung neuer Arten von Klassen (vgl. Kapitel 2.6.1) [51]. Leider konnte "Java TX" aufgrund beschränkter Kapazitäten und anderer Prioritäten, wie bspw. die Erweiterung der Programmiersprache Java um globale Typinferenz, nicht mit diesem Entwicklungstempo mithalten. Aktuell wird jedoch an einer Aktualisierung des "Java TX"-Compilers gearbeitet. Ziel ist es, "Java TX" auf den Stand der aktuellsten LTS-Version von Java (Version 17) anzunähern. Dabei wird der Fokus der Entwickler zunächst auf Funktionen liegen, welche Einfluss auf globale Typinferenz beziehungsweise das Typsystem der Sprache haben. So werden in dieser Arbeit zum Beispiel Neuerungen behandelt, die im Zusammenhang mit dem "Pattern Matching" stehen.

2.6.1 Neue Funktionen im Zusammenhang mit "Pattern Matching"

Das "Pattern Matching" hat seinen Ursprung in funktionalen Programmiersprachen wie "Scala" und "Haskell" und ist dort fester Bestandteil des Funktionsumfangs [45, Kapitel 15] [38, Kapitel 4.4].

Definition 9 *"Pattern Matching": Unter "Pattern Matching" versteht man den Vergleich zwischen einem Wert (bspw. ein Objekt einer Klasse) und einen bestimmten Muster (dem "Pattern"). Passt der Wert auf das Muster, können Teile des Wertes (bspw. eine Subtyp) entsprechend dem Muster für die Weiterverarbeitung extrahiert werden [24].*

In den JDK-Versionen 12–19 wurden verschiedene "Features" in das JDK integriert, die direkt oder indirekt mit "Pattern Matching" zu tun haben. Außerdem beeinflussen sie den Typinferenzalgorihmus, da dieser die neuen Sprachkonstrukte bei der Generierung von Bedingungen und der Inferenz der zugehörigen Typen beachten muss. Somit sind diese Funktionen für die Integration in "Java TX" besonders interessant.

Sealed Classes

"Sealed Classes" und "Sealed Interfaces" erweitern die Programmiersprache Java um Superklassen beziehungsweise Interfaces, die nur bestimmt Vererbungshierarchien zulassen. Damit wird die Nutzung der Superklasse beziehungsweise des Interfaces eingeschränkt. Dadurch bekommen Entwickler bessere Kontrolle darüber, wo die Methoden einer abstrakten

Klasse oder eines Interfaces implementiert werden. Außerdem bilden sie die Basis für eine vollständige Analyse eines "Patterns". Nutzen Entwickler beispielsweise "Pattern Matching" in einem "switch-Statement" (vgl. 2.6.1 auf Seite 43) für eine "Sealed Class" oder ein "Sealed Interface", ist keine Angabe eines "default"-Statements notwendig, wenn alle erlaubten Subtypen überprüft werden (vgl. 2.30 auf Seite 46) [34].

"Sealed Classes" und "Sealed Interfaces" sollen die Deklaration von Beziehungen zwischen Super- und Subtypen vereinfachen und solche Zusammenhänge verständlicher darstellen. Ziel ist es, die Vererbung von Klassen und Interfaces zu beschränken, ohne Subtypen als "final" deklarieren zu müssen oder sie anderweitig zu beschränken. Folgender Quelltext 2.16 zeigt, wie Vererbungshierarchien vor der Einführung beschränkt wurden:

```
1 interface Celestial { ... }
2 final class Planet implements Celestial { ... }
3 final class Star implements Celestial { ... }
4 final class Comet implements Celestial { ... }
```

Quelltext 2.16: Beispiel für die Beschränkung von Vererbung vor "Sealed Classes" und "Sealed Interfaces"
Quelle: [34]

Mit der Einführung von "Sealed Classes" sind solche Deklarationen von Subtypen nicht mehr nötig. Die erlaubten Subtypen können nun direkt angegeben werden (siehe Quelltest 2.17). Dasselbe gilt für "Sealed Interfaces". In "Sealed Classes" gibt es neben der Angabe der erlaubten Subtypen im "Header" der Klasse auch noch die Möglichkeit, die erlaubten Klassen innerhalb der "Sealed Class" zu deklarieren. Der Compiler erstellt dann automatisch die Liste der erlaubten Subtypen [34].

```
1 package com.example.geometry;
2
3 public abstract sealed class Shape permits com.example.polar.Circle,
    ↪ com.example.quad.Rectangle, com.example.quad.simple.Square {
4   ...
5 }
6
7 abstract sealed class Root { ...
8     final class A extends Root { ... }
9     final class B extends Root { ... }
```

```
10      final class C extends Root { ... }
11  }
```

Quelltext 2.17: Beispiele für die Beschränkung von Vererbung durch eine "Sealed Class" Quelle: [34]

Records

Mit "Records" wurde in Java 16 eine neue Art von Klasse eingeführt. Sie erlauben die komfortable Deklaration von einfachen, unveränderlichen Datensammlungen, die häufig in der objektorientierten Programmierung eingesetzt werden. Ein Beispiel für eine solche Datensammlung zeigt Quelltext 2.18. Das Ziel der "Records" ist es, Programmieren die Arbeit zu erleichtern. Der Fokus sollte laut den Entwicklern der "Records" auf der Modellierung von Daten liegen, anstatt auf der Implementierung der immer gleichen Verhaltensweisen solcher Datenstrukturen. Die Implementierung von Konstruktoren, Zugriffsmethoden ("Getter" & "Setter") und Hilfsmethoden, wie "equals", "hashCode" oder "toString", sind fehleranfällig und wenig anspruchsvoll. Außerdem können sie vom Compiler übernommen und automatisiert durchgeführt werden. Dies verhindert auch, dass Entwickler Hilfsmethoden weglassen und dadurch überraschende Verhaltensweisen ihres Programms herbeiführen [32].

```
1  class Point {
2      private final int x;
3      private final int y;
4
5      Point(int x, int y) {
6          this.x = x;
7          this.y = y;
8      }
9
10     int x() { return x; }
11     int y() { return y; }
12
13     public boolean equals(Object o) {
14         if (!(o instanceof Point)) return false;
15         Point other = (Point) o;
16         return other.x == x && other.y == y;
```

```
17      }
18
19      public int hashCode() {
20          return Objects.hash(x, y);
21      }
22
23      public String toString() {
24          return String.format("Point[x=%d,␣y=%d]", x, y);
25      }
26  }
```

Quelltext 2.18: Einfach Datenstruktur mit zwei Variablen, wie sie vor der Einführung von "Records" deklariert werden musste.
Quelle: [32]

Seit dem ersten Vorschlag für "Records", der in Java Version 14 implementiert war, wurden noch einige Änderungen vorgenommen. "Records" erlauben die Deklaration einer Struktur aus Quelltext 2.18 auf der vorherigen Seite in einem kompakten Statement `record Point(int x, int y` Sämtliche Methoden und Felder, die für die Datenstruktur benötigt werden, generiert der Compiler automatisch. Selbstverständlich können im "Block" des "Records" noch weitere Felder, Methoden oder Konstruktoren implementiert werden.

```
1  record Rational(int num, int denom) {
2      Rational(int num, int demon) {
3          int gcd = gcd(num, denom);
4          num /= gcd;
5          denom /= gcd;
6          this.num = num;
7          this.denom = denom;
8      }
9  }
```

Quelltext 2.19: Standardmäßig deklarierter Konstruktor eines Records.
Quelle: [32]

Konstruktoren stellen im Falle von "Records" eine Besonderheit dar. Sie können vorschriftsmäßig mit oder ohne den erwarteten Parametern angegeben werden. Werden die Parameter

angegeben, müssen sie der "Record"-Deklaration entsprechen. Die Konstruktoren in Quelltext 2.19 auf der vorherigen Seite und Quelltext 2.20 sind im Kontext von "Records" identisch [32].

```
record Rational(int num, int denom) {
    Rational {
        int gcd = gcd(num, denom);
        num /= gcd;
        denom /= gcd;
    }
}
```

Quelltext 2.20: Spezieller Konstruktor für "Records" ohne Parameter und Feldzuweisungen. Quelle: [32]

Grundsätzlich verhalten sich "Records" in Java-Programmen wie gewöhnliche Klassen. Allerdings gelten einige Beschränkungen für die Nutzung von "Records". Eine vollständige Auflistung zeigt [32, Abschnitt "Rules for record classes"]

- "Records" können nicht von anderen Klassen erben. Die Superklasse jedes "Records" ist "java.lang.Record"
- Ein "Record" ist automatisch "final" und kann nicht abstrakt sein
- Alle Felder der "Records" sind "final". Es handelt sich um unveränderliche Datensammlungen

"Records" können gut mit "Sealed Classes" kombiniert werden. In Verbindung ergeben sie sogenannte "algebraische Datentypen". So können "Records" beispielsweise für die Implementierung eines "Sealed Interfaces" verwendet werden (siehe Quelltext 2.21). Außerdem eignen sie sich gut für "Pattern Matching", da ihre Schnittstelle an ihren Zustand gekoppelt sind [32].

```
package com.example.expression;

public sealed interface Expr
    permits ConstantExpr, PlusExpr, TimesExpr, NegExpr {...}

public record ConstantExpr(int i) implements Expr {...}
public record PlusExpr(Expr a, Expr b) implements Expr {...}
```

```
8 public record TimesExpr(Expr a, Expr b) implements Expr {...}
9 public record NegExpr(Expr e) implements Expr {...}
```

Quelltext 2.21: Spezieller Konstruktor für "Records" ohne Parameter und Feldzuweisungen. Quelle: [32]

Switch expressions

"Switch Expressions" erweitern die Einsatzmöglichkeiten der "Switch"-Kontrollstruktur. Die "Switch Expression" wurde neben dem bereits existierenden "Switch Statement" in Java Version 14 final eingeführt. Der größte Unterschied zum "Switch Statement" ist die Möglichkeit, eine "Switch Expression" bei der Zuweisung von Variablen oder als Bedingung für Kontrollstrukturen zu verwenden. Dies setzt allerdings voraus, dass eine "Switch Expression" immer einen Wert darstellt beziehungsweise "zurückgibt". Dies ist bei der klassischen Kontrollstruktur, dem "Switch Statement", nicht nötig.

Der Bedarf für "Switch Expressions" ergab sich im Zuge der Vorbereitung der Programmiersprache Java auf die Einführung des "Pattern Matching" (siehe 2.6.1 auf Seite 43). Bei der objektorientierten Programmierung werden Kontrollstrukturen häufig als Ausdrücke benötigt, um beispielsweise den Wert einer Variable an Bedingungen zu knüpfen. Zusätzlich sind klassische "switch"-Statements oft unübersichtlich und es können schnell Fehler entstehen. Dies gilt vor allem für die Definition sogenannter "Fall Throughs", bei denen mehrere "Switch Cases" einem "Block" zugeordnet sind (siehe Quelltext 2.22). Vergisst der Entwickler des Programms ein "break"-Statement, ändert sich der gesamte Programmablauf und der Fehler kann unter Umständen sehr schwer zu finden sein. Außerdem ist bei der Deklaration lokaler Variablen in "Switch Statements" Vorsicht geboten, da alle lokalen Variablen im gesamten "Switch Statement" gültig sind. Dies kann dazu führen, dass aus Versehen Werte in Variablen überschrieben werden und erfordert mehr Deklarationen als nötig (siehe Quelltext 2.22) [9].

```
1 switch (day) {
2     case MONDAY:
3     case TUESDAY:
4         int temp = ... // The scope of 'temp' continues to the }
5         break;
6     case WEDNESDAY:
7     case THURSDAY:
```

```
8           int temp2 = ... // Can't call this variable 'temp'
9           break;
10      default:
11          int temp3 = ... // Can't call this variable 'temp'
12  }
```

Quelltext 2.22: Klassisches "switch"-Statement mit "Fall Throughs"
Quelle: [9]

Deshalb wurden "Switch Expressions", wie in JEP 361 beschrieben, in Java Version 14 als fester Bestandteil der Sprache eingeführt. Als "Expression" können sie beispielsweise auf der rechten Seite einer Zuweisung stehen oder als Parameter in einem Funktionsaufruf verwendet werden. Außerdem nutzen sie sogenannte "Arrow Labels", welche die klassischen "Case Labels" ersetzen können. Der Einsatz von "Arrow Labels" ist auch in "Switch Statements" möglich. Der Unterschied zu den "Case Labels" besteht darin, dass mehrere Fälle in einem einzigen "Label" behandelt werden können. Es müssen keine "Fall Throughs" mehr verwendet werden. Die möglichen Werte können nach dem Keyword "case" aufgelistet werden (siehe Quelltext 2.23).

```
1  int numLetters = switch (day) {
2      case MONDAY, FRIDAY, SUNDAY -> 6;
3      case TUESDAY -> 7;
4      case THURSDAY, SATURDAY -> 8;
5      case WEDNESDAY -> 9;
6  };
```

Quelltext 2.23: Beispiel für eine Zuweisung mit "Switch Expression" und "Arrow Labels" anstatt "Fall Throughs"
Quelle: [9]

Auf der rechten Seite der "Arrow Labels" können Ausdrücke, Statement-Blöcke oder "throw"-Statements (für "Exceptions") verwendet werden. Die Verwendung von Statement-Blöcken löst das Problem des Geltungsbereichs lokaler Variablen. Deklarierte Variablen sind dabei ausschließlich in einem Block gültig und nicht mehr im gesamten "Switch"-Ausdruck. Jedoch entsteht bei der Verwendung von Statement-Blöcken ein neues Problem in Bezug auf die Rückgabe eines Wertes. Steht auf der rechten Seite eines "Arrow Labels" ausschließlich ein Ausdruck, ist klar, welcher Wert zurückgegeben wird. Bei der Verwendung von

Statement-Blöcken bedarf es jedoch einer Möglichkeit die Rückgabe eines Wertes zu initiieren, ähnlich zu "return"-Statements in Funktionen. Hier kommt das neue "yield"-Statement ins Spiel, welches genau diese Aufgabe übernimmt (siehe Quelltext 2.24). Dieses Statement kann auch in "klassischen Case Labels" verwendet werden (siehe Quelltext 2.25) [9].

```
1 int j = switch (day) {
2     case MONDAY -> 0;
3     case TUESDAY -> 1;
4     default -> {
5         int k = day.toString().length();
6         int result = f(k);
7         yield result;
8     }
9 };
```

Quelltext 2.24: "Switch Expression" mit Ausdrücken und Statement-Blöcken
Quelle: [9]

```
1 int result = switch (s) {
2     case "Foo":
3         yield 1;
4     case "Bar":
5         yield 2;
6     default:
7         System.out.println("Neither␣Foo␣nor␣Bar,␣hmmm...");
8         yield 0;
9 };
```

Quelltext 2.25: "Switch Expression" mit "Case Labels" und "yield"-Statements
Quelle: [9]

Pattern Matching

Als Annäherung der objektorientierten Programmiersprache Java an funktionale Sprachen wie "Haskell" und "Scala" wurde in den Java Version 14 bis 21 das "Pattern Matching" eingeführt (siehe Definition 9 auf Seite 36). Es stehen verschiedene Formen von "Patterns"

zur Verfügung. Die klassischen "Type Patterns" bestehen aus einem Typ und einem Bezeichner, unter dem der Zugriff auf das Objekt des Typs erfolgt, sofern der vergangen Vergleich eines Wertes mit dem "Pattern" eine Übereinstimmung ergeben hat. Diese "Type Patterns" können noch um Konditionen zu sogenannten "Guarded Patterns" erweitert werden (siehe Quelltext 2.27 auf der nächsten Seite) [31] [14]. Weiterhin haben Entwickler seit Java Version 19 die Möglichkeit, "Record Patterns" zu verwenden [33]. Sie erlauben die Verwendung von "Records" (siehe 2.6.1 auf Seite 38) für den Vergleich zwischen einem Objekt und einem "Pattern". Die im "Record" enthaltenen Felder können bei erfolgtem Vergleich direkt weiterverwendet werden [13]. Seit Java Version 17 kann "Pattern Matching" in zwei verschiedenen Formen verwendet werden.

"Pattern Matching" mit "instanceof"-Operator Der "instanceof"-Operator ist, wie "&", "|", etc., ein Operator für die Beschreibung von Konditionen. In Verbindung mit "if"-Statements kann er zur Prüfung des Typs eines Objektes verwendet werden. Mit der Einführung des "Pattern Matching" für "instanceof" in Java Version 14 konnten Entwickler zusätzlich zum Typ, den sie prüfen wollten, auch einen Bezeichner angeben. Im nachstehenden Statement-Block der Kontrollstruktur konnte das Objekt dann unter diesem Bezeichner verwendet werden — und zwar mit dem Typ aus dem "Pattern". Die Typumwandlung muss nicht mehr manuell vorgenommen werden, sondern erfolgt automatisch [29]. So kann die Anzahl expliziter Typumwandlungen reduziert werden [31]. Diese Veränderung ist in Quelltext 2.26 dargestellt.

```
1 if (obj instanceof String) {
2     String s = (String) obj;
3     System.out.println(s);
4 }
5
6 if (obj instanceof String s) {
7   System.out.println(s);
8 }
```

Quelltext 2.26: Veränderung des "instanceof"-Operators durch "Pattern Matching"
Quelle: [31]

Selbstverständlich können auch "Guarded Patterns" mit dem "instanceof"-Operator verwendet werden. Ein Beispiel dafür zeigt das "if"-Statement in folgendem Quelltext 2.27. Die finale Version des "Pattern Matching" mit "instanceof" wurde mit Java Version 21 veröffentlicht [31].

```
1 if (obj instanceof String s && s.length() > 5) {
2     flag = s.contains("jdk");
3 }
```

Quelltext 2.27: "Guarded Patterns" mit "instanceof"-Operator
Quelle: [31]

"Pattern Matching" in "Switch"-Konstrukten Möchte man mehrere Typen mit einem Objekt abgleichen, entstehen bei der Verwendung des "instanceof"-Operators schnell große, unübersichtliche "if-else"-Strukturen wie in Quelltext 2.28. Seit Java Version 17 können diese Strukturen nun auch mittels kompakten "Switch"-Konstrukten erstellt werden [10].

```
1  static String formatter(Object obj) {
2      String formatted = "unknown";
3      if (obj instanceof Integer i) {
4          formatted = String.format("int␣%d", i);
5      } else if (obj instanceof Long l) {
6          formatted = String.format("long␣%d", l);
7      } else if (obj instanceof Double d) {
8          formatted = String.format("double␣%f", d);
9      } else if (obj instanceof String s) {
10         formatted = String.format("String␣%s", s);
11     }
12     return formatted;
13   }
```

Quelltext 2.28: "if-else"-Konstrukt mit mehreren "instanceof"-Vergleichen
Quelle: [14]

Dazu gibt der Entwickler "Patterns" in den zugehörigen "Labels" an. Stimmt der Typ des betreffenden Objekts mit dem Typ des "Patterns" überein, kann das konvertierte Objekt unter dem angegeben Bezeichner im Ausdruck beziehungsweise im Statement-Block des

"Switch Cases" verwendet werden (siehe Quelltext 2.29). Mit der Verwendung von "Switch Expressions" können damit Ausdrücke geschaffen werden, die den Kontrollstrukturen funktionaler Programmiersprachen, wie "Haskell" oder "Scala", ähneln [38, Kapitel 4.4] [27, Kapitel 1.6]. Neben den "Type Patterns" können auch "Guarded Patterns" und "Record Patterns" verwendet werden. Diese können sogar geschachtelt werden, um Typen und enthaltene Daten noch besser verwalten zu können (siehe Beispiel in 2.31 auf der nächsten Seite) [13].

```
1 static String formatterPatternSwitch(Object obj) {
2     return switch (obj) {
3         case Integer i -> String.format("int␣%d", i);
4         case Long l -> String.format("long␣%d", l);
5         case Double d -> String.format("double␣%f", d);
6         case String s -> String.format("String␣%s", s);
7         default -> obj.toString();
8     };
9 }
```

Quelltext 2.29: Kompaktere Darstellung des Quelltextes 2.28 auf der vorherigen Seite mit "Pattern Matching" in "Switch Expression"
Quelle: [14]

Wird überprüft, ob es sich bei einem Objekt um eine "Sealed Class" handelt (vgl. Kapitel 2.6.1 auf Seite 36), ist keine Angabe eines "Default Cases" in der "Switch"-Struktur notwendig (vgl. Quelltext 2.30). Der Compiler kennt alle erlaubten Subtypen der "Sealed Class" und erwartet daher keine Anweisungen für den Fall, dass kein "Case" zutrifft. Dafür generiert er aber eine Fehlermeldung, wenn nicht alle erlaubten Subtypen der "Sealed Class" durch die "Switch"-Struktur abgedeckt sind.

```
1 Shape rotate(Shape shape, double angle) {
2     return switch (shape) { // pattern matching switch
3         case Circle c -> c;
4         case Rectangle r -> shape.rotate(angle);
5         case Square s -> shape.rotate(angle);
6         // no default needed!
7     }
```

```
}
```

Quelltext 2.30: "Pattern Matching" in "Switch"-Konstrukten für "Sealed Classes"
Quelle: [34]

```
record Point(int x, int y) {}
enum Color { RED, GREEN, BLUE }
interface Shape {}
record ColoredPoint(Point pt, Color color) {}
record Rectangle(ColoredPoint upperLeft, ColoredPoint lowerRight)
    ↪ implements Shape {}

class PatternMatching {
  void printColorOfUpperLeftPoint(Shape shape)
  {
    switch (shape) {
      case Rectangle(ColoredPoint(Point pt, Color color),
          ↪ ColoredPoint lowerRight) -> System.out.println("x:␣" + pt
          ↪ .x() + "␣/␣color:␣" + color + "␣/␣lowerRight:␣" +
          ↪ lowerRight);

      default -> System.out.println("not␣a␣rectangle");
    }
  }
}
```

Quelltext 2.31: Geschachtelte "Record Patterns" in "Switch"-Konstrukt

3 Aktualisierung des “Java TX”-Compilers

Zur Umsetzung der vorgesehenen Aktualisierungen in “Java TX” werden sämtliche, am Kompilierungsprozess beteiligten, Komponenten angepasst und erweitert (vgl. Abschnitt 2.5.4 auf Seite 33). Vor Beginn der vorliegenden Arbeit wurde die Grammatik des Compilers bereits aktualisiert. Der “Antlr Parser” ist auf dem Stand der Java Version 17 und kann alle syntaktischen Elemente dieser Version verarbeiten und im “Parsetree” darstellen (vgl. Kapitel 2.2.2 auf Seite 13). Auch der “SyntaxTreeGenerator” enthält bereits erste Funktionen zur Umwandlung der neuen “Parsetrees” in AST (vgl. Kapitel 2.4.1 auf Seite 22). Jedoch müssen zunächst neue Klassen für den AST erstellt werden, um die neuen Sprachkonstrukte in diesem repräsentieren zu können (siehe Abschnitt 3.2 auf Seite 50). Danach werden “Unit-Tests” für den aktualisierten “SyntaxTreeGenerator” erstellt, um dessen korrekte Funktionsweise auch nach weiteren Anpassungen des Compilers überprüfen zu können (siehe Abschnitt 3.3 auf Seite 65).

3.1 Projektstruktur & Tools

Der “Java TX”-Compiler ist in der Programmiersprache Java implementiert [49]. Aktuell kommt für die Implementierung Java Version 20 zum Einsatz. Für die Verwaltung externer Bibliotheken, der Konfiguration und für automatisierte Kompilierung und Testabläufe wird das Werkzeug “Maven” verwendet. Das populäre “Build Tool” kommt häufig in Java-Projekten zum Einsatz [8, Kapitel 1]. Die Definition der externen Bibliotheken und der Konfigurationsparameter des Projekts erfolgt in einer speziellen XML-Datei, “pom.xml”. Sie liegt im obersten Verzeichnis des Projektes. Der Quelltext des “Java TX”-Compilers befindet sich im Verzeichnis “src/main”. Alle Klassen für Tests sind im Verzeichnis “src/test” enthalten. Der Ordner “target” enthält die kompilierten Binärdateien für den “Java TX”-Compiler. Somit entspricht die Projektstruktur den Vorgabe für “Maven-Projekte” [8, Kapitel 1]. Einzig der Ordner “resources” im Projektverzeichnis passt nicht in diese Struktur. Er enthält “Java TX”-Quelltext in Dateien mit der Endung “.jav” sowie JVM-Bytecode zur Durchführung von “Unit-Tests”. Zusätzlich gibt es im Hauptverzeichnis noch Ordner

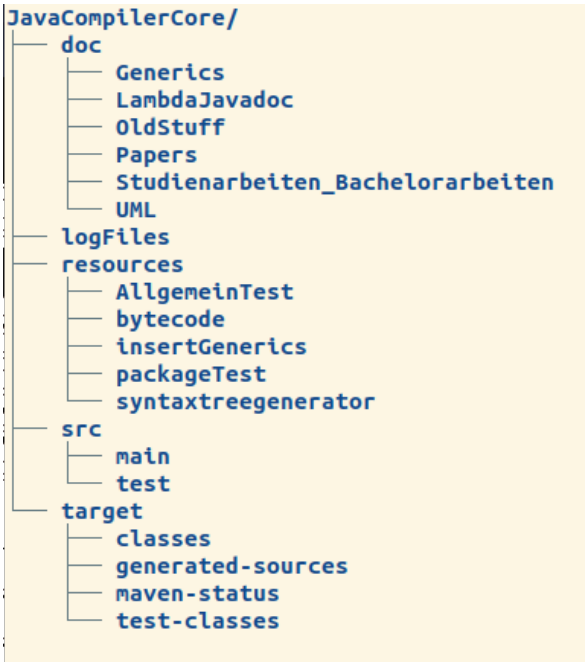


Abbildung 3.1: Ordnerstruktur des Hauptverzeichnnisses für das Projekt "JavaCompilerCore" zur Entwicklung des "Java TX"-Compilers

für die Dokumentation des Compilers sowie einen Ordner für Logdateien. Die vollständige Projektstruktur ist in Anhang C auf Seite 77 abgebildet. Dort sind auch die in Kapitel 2.5.4 auf Seite 33 enthaltenen Komponenten zu erkennen. Im folgenden erfolgen alle Angaben von Paketnamen relativ zu "de.dhbwstuttgart".

3.1.1 Versionsverwaltung

Der Quellcode des "Java TX"-Compilers wird mit dem Versionsverwaltungswerkzeug "git" verwaltet. Es existieren verschiedene "Branches", auf denen zurzeit am Compiler gearbeitet wird. Der Vorteil von "git" gegenüber anderen Versionsverwaltungstools, wie zum Beispiel "SVN", besteht darin, dass Entwickler den Quelltext lokal bearbeiten können. Diese Änderungen werden in Form von "Committs" festgeschrieben. Anschließend kann ein Abgleich mit der Version, die auf einem zentralen Server gespeichert ist, erfolgen. Dies führt zu weniger aufwendigen Synchronisationsoperationen als sie beispielsweise bei "SVN" auftreten, wo jede Veränderung direkt auf den zentralen Server "committet" wird. "Git" lässt sich auch ausschließlich auf dem lokalen Rechner verwenden, ohne dass ein Server involviert ist.

Die aktuelle "Hauptbranch" des "Java TX"-Compilers heißt "bigRefactoring". Im "Branch" "targetBytecode" wird am "Backend" des Compilers gearbeitet. Die hier beschriebenen Anpassungen werden in der "Branch" "patternMatching" vorgenommen. Diese beiden "Branches" werden über sogenannte "Merges" in der "Hauptbranch" miteinander synchronisiert.

3.2 Erweiterung des AST

Der AST bildet die Grundlage für die Generierung von ausführbarem Maschinencode bzw. Bytecode. Deshalb müssen die neuen Sprachkonstrukte aus Java in den AST von "Java TX" aufgenommen werden. Dazu werden Klassen im Paket "syntaxtree" erstellt (siehe Projektstruktur in Anhang C auf Seite 77). Jede dieser Klassen repräsentiert eine Struktur zur Verwendung in "Java TX"-Programmen. Die Klassen dienen hauptsächlich der Speicherung von Informationen und bestehen daher größtenteils aus Feldern und "Getter"- bzw. "Setter"-Methoden. Sie werden, sobald das "Backend" des Compilers aktiviert wird, von einem sogenannten "ASTVisitor" ausgelesen, um den AST in JVM-Bytecode umzuwandeln (siehe Kapitel 2.1 auf Seite 6).

3.2.1 Records

Da "Records" gewöhnliche Klassen darstellen, wurde der neue Typ "Record" im AST von "Java TX" von der vorhandenen Klasse "ClassOrInterface" abgeleitet. Die Inhalte der beiden Klassen sind identisch (vgl. Anhang E.1 auf Seite 86). Die Vererbung dient der Identifikation von "Records" im AST (beispielsweise durch "Pattern Matching"). Außerdem wurde der "SyntaxTreeGenerator" angepasst. Trifft dieser auf eine "Record"-Deklaration im "Parsetree", wird daraus automatisch ein Knoten vom Typ "Record" abgeleitet. Der genaue Programmablauf ist in Quelltext 3.1 dargestellt.

```
1 private de.dhbwstuttgart.syntaxtree.Record convertRecord(
    ↪ RecordDeclarationContext recordDeclaration, int modifiers) {
2     String identifier = recordDeclaration.identifier().getText();
3     String className = this.pkgName + (this.pkgName.length() > 0 ? "."
        ↪ : "") + identifier;
```

```
4     JavaClassName name = reg.getName(className); // Holt den Package
          ↪ Namen mit dazu
5     Token offset = recordDeclaration.getStart();
6     GenericsRegistry generics = createGenerics(recordDeclaration.
          ↪ genericDeclarationList(), name, "", reg, new GenericsRegistry
          ↪ (globalGenerics));
7     if (!name.toString().equals(className)) { // Kommt die Klasse schon
          ↪  in einem anderen Package vor?
8       throw new TypeinferenceException("Name␣" + className + "␣bereits␣
            ↪ vorhanden␣in␣" + reg.getName(className).toString(),
            ↪ recordDeclaration.getStart());
9     }
10    GenericDeclarationList genericClassParameters;
11    if (recordDeclaration.genericDeclarationList() == null) {
12      genericClassParameters = new GenericDeclarationList(new ArrayList
            ↪ <>(), recordDeclaration.identifier().getStop());
13    } else {
14      genericClassParameters = TypeGenerator.convert(recordDeclaration.
            ↪ genericDeclarationList(), name, "", reg, generics);
15    }
16    RefType superClass = new RefType(ASTFactory.createObjectClass().
          ↪ getClassName(), offset);
17    List<Field> fielddecl = new ArrayList<>();
18    List<Method> methods = new ArrayList<>();
19    List<Constructor> constructors = new ArrayList<>();
20    Boolean isInterface = false;
21    List<RefType> implementedInterfaces = new ArrayList<>();
22    List<FormalParameter> constructorParameters = new ArrayList<>();
23    List<Statement> constructorStatements = new ArrayList<>();
24    for (RecordComponentContext component : recordDeclaration.
          ↪ recordHeader().recordComponentList().recordComponent()) {
25      int fieldmodifiers = allmodifiers.get("private") + allmodifiers.
            ↪ get("final");
26      String fieldname = component.identifier().getText();
27      Token fieldoffset = component.getStart();
```

```
28        RefTypeOrTPHOrWildcardOrGeneric fieldtype = null;
29        if (Objects.isNull(component.typeType())) {
30          fieldtype = TypePlaceholder.fresh(offset);
31        } else {
32          fieldtype = TypeGenerator.convert(component.typeType(), reg,
              ↪ generics);
33        }
34        fielddecl.add(new Field(fieldname, fieldtype, fieldmodifiers,
              ↪ fieldoffset));
35        constructorParameters.add(new FormalParameter(fieldname,
              ↪ fieldtype, fieldoffset));
36        FieldVar fieldvar = new FieldVar(new This(offset), fieldname,
              ↪ fieldtype, fieldoffset);
37        constructorStatements.add(new Assign(new AssignToField(fieldvar),
              ↪  new LocalVar(fieldname, fieldtype, fieldoffset), offset));
38        Statement returnStatement = new Return(fieldvar, offset);
39        methods.add(new Method(allmodifiers.get("public"), fieldname,
              ↪ fieldtype, new ParameterList(new ArrayList<>(), offset),
              ↪ new Block(Arrays.asList(returnStatement), offset), new
              ↪ GenericDeclarationList(new ArrayList<>(), offset), offset))
              ↪ ;
40      }
41      RefType classType = ClassOrInterface.generateTypeOfClass(reg.
            ↪ getName(className), genericClassParameters, offset);
42      Constructor implicitConstructor = new Constructor(allmodifiers.get(
            ↪ "public"), identifier, classType, new ParameterList(
            ↪ constructorParameters, offset), new Block(
            ↪ constructorStatements, offset), genericClassParameters,
            ↪ offset);
43      Optional<Constructor> initializations = Optional.of(
            ↪ implicitConstructor);
44      constructors.add(implicitConstructor);
45      for (ClassBodyDeclarationContext bodyDeclaration :
            ↪ recordDeclaration.recordBody().classBodyDeclaration()) {
```

```
46        convert(bodyDeclaration, fielddecl, constructors, methods, name,
              ↪ superClass, generics);
47      }
48      if (!Objects.isNull(recordDeclaration.IMPLEMENTS())) {
49        implementedInterfaces.addAll(convert(recordDeclaration.typeList()
              ↪ , generics));
50      }
51      return new Record(modifiers, name, fielddecl, initializations,
            ↪ methods, constructors, genericClassParameters, superClass,
            ↪ isInterface, implementedInterfaces, offset);
52    }
```

Quelltext 3.1: Methode zur Umwandlung eines "RecordDeclarationContext" aus dem "Parsetree" in eine Klasse im AST

Für jeden Parameter (hier "RecordComponentContext") des "Records" wird ein Feld erstellt. Diese werden als "private" deklariert, sodass der Zugriff auf die Daten nur mit Methoden erfolgen kann. Diese sogenannten "Getter"-Methoden zum Auslesen der Daten aus einem "Record" werden ebenfalls generiert und der Klasse hinzugefügt. Dasselbe gilt für den Konstruktor der Klasse, der ebenfalls auf Basis der "Record"-Deklaration generiert werden kann (vgl. Kapitel 2.6.1 auf Seite 38 und "for"-Schleife ab Zeile 24 in Quelltext 3.1 auf Seite 50). Sind alle Bestandteile der Klasse erstellt, wird eine neue "Record"-Instanz erstellt (Zeile 51 in Quelltext 3.1 auf Seite 50). Dazu kommt der in Quelltext 3.2 dargestellte Konstruktor der Superklasse zum Einsatz.

```
1 public class ClassOrInterface extends SyntaxTreeNode implements
    ↪ TypeScope {
2   private Boolean methodAdded = false; // wird benoetigt bei in
      ↪ JavaTXCompiler.getConstraints()
3   protected int modifiers;
4   protected JavaClassName name;
5   private List<Field> fields = new ArrayList<>();
6   private Optional<Constructor> fieldInitializations; // PL 2018-11-24:
      ↪  Noetig, um Bytecode fuer initializators nur einmal zu erzeugen
7   private List<Method> methods = new ArrayList<>();
8   private GenericDeclarationList genericClassParameters;
9   private RefType superClass;
```

```
10   protected boolean isInterface;
11   private List<RefType> implementedInterfaces;
12   private List<RefType> permittedSubtypes;
13   private List<Constructor> constructors;
14
15   public ClassOrInterface(int modifiers, JavaClassName name, List<Field
        ↪ > fielddecl, Optional<Constructor> fieldInitializations, List<
        ↪ Method> methods, List<Constructor> constructors,
        ↪ GenericDeclarationList genericClassParameters, RefType
        ↪ superClass, Boolean isInterface, List<RefType>
        ↪ implementedInterfaces, List<RefType> permittedSubtypes, Token
        ↪ offset) {
16     super(offset);
17     if (isInterface && !Modifier.isInterface(modifiers))
18       modifiers += Modifier.INTERFACE;
19     this.modifiers = modifiers;
20     this.name = name;
21     this.fields = fielddecl;
22     this.fieldInitializations = fieldInitializations;
23     this.genericClassParameters = genericClassParameters;
24     this.superClass = superClass;
25     this.isInterface = isInterface;
26     this.implementedInterfaces = implementedInterfaces;
27     this.permittedSubtypes = permittedSubtypes;
28     this.methods = methods;
29     this.constructors = constructors;
30   }
31
32 [...]
```

Quelltext 3.2: Auszug aus der Klasse "ClassOrInterface" im Paket "syntaxtree"

3.2.2 "instanceof"-Operator

Für den "instanceof"-Operator diente die vorhanden Klasse "BinaryExpr", aus dem Paket "syntaxtree.statement" (vgl. C auf Seite 77), als Vorlage. Sie besitzt dieselbe syntaktische Form und eignet sich daher als "Superklasse" des neuen Operators. Die Bestandteile des Ausdrucks `lexpr instanceof rexpr` werden in den Feldern "lexpr", "rexpr" und "operator" gespeichert. Diese Felder erbt die Klasse "Instanceof" von "BinaryExpr" (siehe Quelltext 3.3). Zusätzlich wird in der Klasse "Instanceof" ein weiteres Feld eingeführt, welches Informationen über den Typ bzw. das "Pattern" auf der rechten Seite des Ausdrucks enthält. Das Feld "pattern" enthält den Typ, mit dem der Typ des Ausdrucks in "lexpr" verglichen werden soll. Dafür kommt eine Instanz der Klasse "Pattern" zum Einsatz (vgl. Kapitel 3.2.3 auf Seite 58 und Anhang E.3 auf Seite 88). Hat der Entwickler kein "Pattern", sondern nur einen Typ, angegeben, wird das Feld "name" des "Pattern"-Objekts mit `null` initialisiert (siehe Quelltext 3.4 auf der nächsten Seite und Quelltext E.2 auf Seite 87).

```
1  public class BinaryExpr extends Expression {
2
3    [...]
4
5    public enum Operator {
6        ADD, // +
7        SUB, // -
8        MUL, // *
9        MOD, // Modulo Operator %
10       AND, // &
11       OR, // |
12       DIV, // /
13       LESSTHAN, // <
14       BIGGERTHAN, // >
15       LESSEQUAL, // <=
16       BIGGEREQUAL, // >=
17       EQUAL, // ==
18       NOTEQUAL, // !=
19       INSTOF // instanceof
20   }
21
```

```
22      public final Operator operation;
23      public final Expression lexpr;
24      public final Expression rexpr;
25
26      public BinaryExpr(Operator operation,
            ↪ RefTypeOrTPHOrWildcardOrGeneric type, Expression lexpr,
            ↪ Expression rexpr, Token offset) {
27          super(type, offset);
28
29          this.operation = operation;
30          this.lexpr = lexpr;
31          this.rexpr = rexpr;
32      }
33  }
```

Quelltext 3.3: Felder und Konstruktor der Klasse "BinaryExpr" aus "syntaxtree.statement"

```
1   public class InstanceOf extends BinaryExpr {
2     private Pattern pattern;
3
4     public InstanceOf(Expression expr, RefTypeOrTPHOrWildcardOrGeneric
          ↪ reftype, Token offset) {
5         super(BinaryExpr.Operator.INSTOF, TypePlaceholder.fresh(offset)
              ↪ , expr, new LocalVar("", reftype, reftype.getOffset()),
              ↪ offset);
6         this.pattern = new Pattern(null, reftype, offset);
7     }
8
9     public InstanceOf(Expression expr, Pattern pattern, Token offset) {
10        super(BinaryExpr.Operator.INSTOF, TypePlaceholder.fresh(offset)
              ↪ , expr, new LocalVar(pattern.getName(), pattern.getType()
              ↪ , pattern.getOffset()), offset);
11        this.pattern = pattern;
12    }
13
14    [...]
```

```
15  }
```

Quelltext 3.4: Konstruktoren der Klasse "Instanceof" zur Darstellung von Ausdrücken mit dem Operator im AST

Für die Umwandlung von Ausdrücken mit dem "instanceof"-Operator aus dem "Parsetree" zu Knoten im AST wird der "StatementGenerator" verwendet. Er wird vom "SyntaxTreeGenerator" instantiiert und für die Umwandlung von Statements und Ausdrücken aufgerufen. Quelltext 3.5 zeigt den Umwandlungsprozess im "StatementGenerator". Die Methode nutzt "Pattern Matching" in "Switch"-Statements, um eventuell verwendete "Patterns" zu analysieren (siehe Kapitel 2.6.1 auf Seite 43).

```
1   [...]
2
3   private Expression convert(Java17Parser.InstanceofexpressionContext
        ↪ expression) {
4     Expression left = convert(expression.expression());
5     Token offset = expression.getStart();
6     if (Objects.isNull(expression.pattern())) {
7       return new InstanceOf(left, TypeGenerator.convert(expression.
            ↪ typeType(), reg, generics), offset);
8     } else {
9       switch (expression.pattern()) {
10        case PPatternContext primaryPattern:
11        switch (primaryPattern.primaryPattern()) {
12          case TPatternContext typePattern:
13          TypePatternContext typePatternCtx = typePattern.typePattern()
                ↪ ;
14          String localVarName = typePatternCtx.identifier().getText();
15          RefTypeOrTPHOrWildcardOrGeneric localVarType = TypeGenerator.
                ↪ convert(typePatternCtx.typeType(), reg, generics);
16          localVars.put(localVarName, localVarType);
17          return new InstanceOf(left, new Pattern(localVarName,
                ↪ localVarType, typePatternCtx.getStart()), offset);
18          default:
19          throw new NotImplementedException();
20        }
```

```
21          default:
22          throw new NotImplementedException();
23        }
24      }
25    }
26
27    [...]
```

Quelltext 3.5: Umwandlung von "instanceof"-Ausdrücken im "Parsetree" zu AST-Knoten

3.2.3 Pattern

Für die Einführung von "Pattern Matching" in "Java TX" wurden drei neuen Knotentypen in den AST eingefügt. Die Klasse "Pattern" repräsentiert "Type Pattern" im AST. "GuardedPattern" und "RecordPattern" sind für die Darstellung der gleichnamigen "Pattern" verantwortlich (siehe Kapitel 2.6.1 auf Seite 43). Abbildung 3.2 zeigt die Vererbungsstruktur der "Pattern"-Knoten im Paket "syntaxtree.statement".

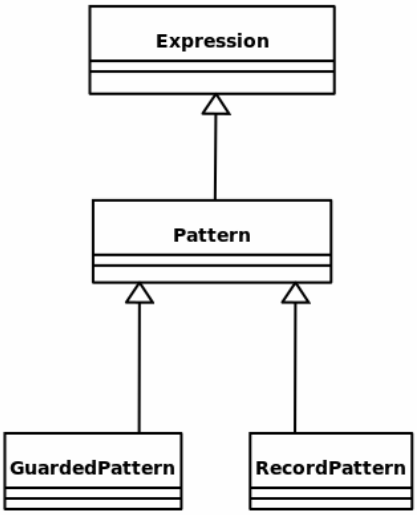


Abbildung 3.2: Vererbungshierarchie die "Pattern"-Klassen

"Pattern" Die Klasse "Pattern" enthält, zusätzlich zum Feld "type" der Klasse "Expression", das Feld "name". Dieses Feld enthält den Bezeichner mit dem auf das "Pattern"

zugegriffen werden kann, sofern der Vergleich mit dem zu untersuchenden Objekt erfolgreich war (siehe Kapitel 2.6.1 auf Seite 43).

```
1  public class Pattern extends Expression {
2
3    private String name;
4
5    public Pattern(String name, RefTypeOrTPHOrWildcardOrGeneric type,
       ↪ Token offset) {
6      super(type, offset);
7      this.name = name;
8    }
9
10   [...]
11 }
```

Quelltext 3.6: Feld und Konstruktor der Klasse "Pattern"

"GuardedPattern" Die Subklasse "GuardedPattern" erweitert "Pattern" um ein zusätzliches Feld, das eine Sammlung möglicher Konditionen enthält. Diese Konditionen sind als Instanzen der Klasse "Expression" hinterlegt und in einer Liste gespeichert (siehe Quelltext 3.7).

```
1  public class GuardedPattern extends Pattern {
2
3    private List<Expression> conditions;
4
5    public GuardedPattern(List<Expression> conditions, String name,
       ↪ RefTypeOrTPHOrWildcardOrGeneric type, Token offset) {
6      super(name, type, offset);
7      this.conditions = conditions;
8    }
9
10   [...]
11 }
```

Quelltext 3.7: Feld und Konstruktor der Klasse "GuardedPattern"

"RecordPattern" Die zweite Variante des "Type Pattern", das "Record Pattern", enthält eine Sammlung sogenannter "Subpattern", anstelle der Konditionen im "GuardedPattern". Sie dient der Speicherung von geschachtelten "Pattern" wie sie in Kapitel 2.6.1 auf Seite 43 beschrieben wurden. Jedes "Subpattern" ist eine Instanz der Superklasse "Pattern". Als Datenstruktur kommt hier eine Liste vom Typ "java.util.ArrayList", zum Einsatz. Sie wird für jede Instanz der Klasse initialisiert. Hat das "Record Pattern" keine "Subpattern", bleibt die Liste leer (siehe Quelltext 3.8).

```
1  public class RecordPattern extends Pattern {
2
3    private List<Pattern> subPattern = new ArrayList<>();
4
5    public RecordPattern(String name, RefTypeOrTPHOrWildcardOrGeneric
       ↪ type, Token offset) {
6      super(name, type, offset);
7    }
8
9    public RecordPattern(List<Pattern> subPattern, String name,
       ↪ RefTypeOrTPHOrWildcardOrGeneric type, Token offset) {
10     super(name, type, offset);
11     this.subPattern = subPattern;
12   }
13
14   [...]
15 }
```

Quelltext 3.8: Feld und Konstruktor der Klasse "RecordPattern"

Erstellung der "Pattern-Knoten" Auch für die Erstellung der "Pattern" ist der "StatementGenerator" verantwortlich. Für die Bestimmung der Art des vorliegenden "Patterns" kommen "switch"-Strukturen mit "Pattern Matching" zum Einsatz. Der "Knoten" des "Parsetrees" wird, mithilfe von "Labels", analysiert. Auf Basis des Ergebnisses wird dann eine Instanz der entsprechenden "Pattern"-Klasse erzeugt. Die kontextfreie Grammatik des "Parsers" gruppiert "Type-" und "Record Pattern" zu sogenannten "Primary Pattern". Diese werden von den "Guarded Pattern" unterschieden. Deswegen ist die Umwandlung eines

"PatternContext" (Knoten im "Parsetree") auf drei Methoden aufgeteilt. Quelltext 3.9 zeigt diese drei Methoden. Die Methode `convert(PatternContext pattern)` wird während der Umwandlung von "Switch Statements" und "Switch Expressions" aufgerufen.

```
1   [...]
2
3   private Pattern convert(PatternContext pattern) {
4     return switch (pattern) {
5       case PPatternContext pPattern -> {
6         yield convert(pPattern.primaryPattern());
7       }
8       case GPatternContext gPattern -> {
9         GuardedPatternContext guarded = gPattern.guardedPattern();
10        List<Expression> conditions = guarded.expression().stream().map
              ↪ ((expr) -> {
11          return convert(expr);
12        }).toList();
13        yield new GuardedPattern(conditions, guarded.identifier().
              ↪ getText(), TypeGenerator.convert(guarded.typeType(), reg,
              ↪  generics), guarded.getStart());
14      }
15      default -> throw new NotImplementedException();
16    };
17  }
18
19  private Pattern convert(PrimaryPatternContext pPattern) {
20    switch (pPattern) {
21      case TPatternContext tPattern:
22      TypePatternContext typePattern = tPattern.typePattern();
23      return new Pattern(typePattern.identifier().getText(),
            ↪ TypeGenerator.convert(typePattern.typeType(), reg, generics
            ↪ ), typePattern.getStart());
24      case RPatternContext rPattern:
25      RecordPatternContext recordPattern = rPattern.recordPattern();
26      return convert(recordPattern);
27      default:
```

```
28        throw new NotImplementedException();
29
30      }
31    }
32
33    private RecordPattern convert(RecordPatternContext recordPatternCtx)
          ↪ {
34      List<PatternContext> subPatternCtx = recordPatternCtx.
          ↪ recordStructurePattern().recordComponentPatternList().pattern
          ↪ ();
35      List<Pattern> subPattern = subPatternCtx.stream().map((patternCtx)
          ↪ -> {
36        return convert(patternCtx);
37      }).collect(Collectors.toList());
38      IdentifierContext identifierCtx = recordPatternCtx.identifier();
39      return new RecordPattern(subPattern, (identifierCtx != null) ?
          ↪ identifierCtx.getText() : null, TypeGenerator.convert(
          ↪ recordPatternCtx.typeType(), reg, generics), recordPatternCtx
          ↪ .getStart());
40    }
41
42    [...]
```

Quelltext 3.9: Methoden zur Umwandlung von "Pattern" im "Parsetree" zu entsprechenden Knoten im AST

3.2.4 "switch"-Konstrukte

Zur Realisierung von "Pattern Matching" in "switch"-Konstrukten in "Java TX" wurden drei neue Klassen ins Paket "syntaxtree.statement" aufgenommen. Da die gewöhnlichen "switch"-Statements zu Beginn dieser Arbeit noch nicht in "Java TX" integriert waren, wurde auch deren Nutzung durch die neuen Klassen ermöglicht. Die Unterscheidung zwischen "switch"-Statement und "switch-Expression" erfolgt über das Feld "isStatement" der Klasse "Statement", von der die Klasse "Switch" abgeleitet ist. Ruft man die Methode `getStatement()` einer "Switch"-Instanz auf, erhält man einen "Boolean"-Wert, der an-

gibt, ob es sich um eine ("switch"-) "Expression" oder um ein ("switch"-) Statement handelt. Zusätzlich werden in der Klasse "Switch" zwei weitere Felder deklariert, die Informationen über die betreffende "switch"-Struktur enthalten (vgl. Quelltext 3.10 und Anhang E.6 auf Seite 91).

```
1  public class Switch extends Statement {
2
3    private Expression switchedExpression;
4    private List<SwitchBlock> blocks = new ArrayList<>();
5
6    public Switch(Expression switched, List<SwitchBlock> blocks,
       ↪ RefTypeOrTPHOrWildcardOrGeneric type, Boolean isStatement,
       ↪ Token offset) {
7      super(type, offset);
8      if (isStatement)
9        setStatement();
10     this.switchedExpression = switched;
11     this.blocks = blocks;
12   }
13
14   [...]
15 }
```

Quelltext 3.10: Felder und Konstruktor der Klasse "Switch", welche "switch"-Konstrukte im AST repräsentiert

Die "switchedExpression" enthält den Ausdruck für die Vergleiche in den verschiedenen "Labels". Die Liste vom Typ "java.util.ArrayList" enthält Instanzen der Klasse "SwitchBlock", welche wiederum die zugehörigen "Switch Labels" und Statements enthalten. Die Statements werden im Feld "statements" gespeichert, welches die Klasse "SwitchBlock" von "Block" erbt (vgl. Quelltext 3.11 auf der nächsten Seite und Anhang E.7 auf Seite 92). Die Klasse "SwitchLabel" erbt von der Klasse "Expression" und erweitert diese um eine "caseExpression" und einen "Boolean"-Wert, der angibt, ob es sich um den "Default Case" handelt (vgl. Kapitel 2.6.1 auf Seite 41, Quelltext 3.12 auf der nächsten Seite und Anhang E.8 auf Seite 94).

```
1  public class SwitchBlock extends Block {
2
3    private List<SwitchLabel> labels = new ArrayList<>();
4
5    private Boolean defaultBlock = false;
6
7    public SwitchBlock(List<SwitchLabel> labels, Block statements,
       ↪ Token offset) {
8      super(statements.getStatements(), offset);
9      this.labels = labels;
10   }
11
12   public SwitchBlock(List<SwitchLabel> labels, Block statements,
       ↪ Boolean isDefault, Token offset) {
13     super(statements.getStatements(), offset);
14     this.labels = labels;
15     this.defaultBlock = isDefault;
16   }
17
18   [...]
19 }
```

Quelltext 3.11: Felder und Konstruktor der Klasse "SwitchBlock"

```
1  public class SwitchLabel extends Expression {
2
3    private Expression caseExpression;
4    private Boolean defaultCase = false;
5
6    public SwitchLabel(Expression caseExpression,
       ↪ RefTypeOrTPHOrWildcardOrGeneric type, Token offset) {
7      super(type, offset);
8      this.caseExpression = caseExpression;
9    }
10
11   public SwitchLabel(RefTypeOrTPHOrWildcardOrGeneric type, Token
```

```
        ↪ offset) {
12        super(type, offset);
13        this.defaultCase = true;
14      }
15
16      [...]
17    }
```

Quelltext 3.12: Felder und Konstruktor der Klasse "SwitchLabel"

3.3 Unit-Tests für "Syntaxtreegenerator"

Der AST eines "Java TX"-Programms wird durch Umwandlung des "Parsetrees" generiert (siehe 2.4.1 auf Seite 22) und repräsentiert das Programm gegenüber Komponenten, welche folgende Schritte im Kompilierungsprozess übernehmen (wie zum Beispiel der Typinferenzalgorithmus oder der "Bytecodegenerator", vgl. Kapitel 2.5.4 auf Seite 33). Diese Aufgabe übernimmt der "SyntaxTreeGenerator" in Zusammenarbeit mit weiteren Komponenten aus dem Paket "parser.SyntaxTreeGenerator". Zusammen bilden sie eine sogenannte "Unit", deren korrekte Funktionsweise mittels "Unit-Tests" sichergestellt werden soll. Dadurch soll verhindert werden, dass es durch Veränderungen am Quellcode des "Java TX"-Compilers zu Programmfehlern kommt. Das Testen ist ein fester Bestandteil der Softwareentwicklung und trägt maßgeblich zur Qualitätssicherung eines Programms bei [41, Kapitel 1].

Das Werkzeug "Maven", dass in Kapitel 3.1 auf Seite 48 vorgestellt wurde, kann sämtliche Tests automatisch ausführen. Änderungen am Quellcode sollten nur übernommen werden, sofern es bei den "Java TX"-Tests nicht zu Fehlern kommt. Die Tests des "Java TX"-Compilers werden mithilfe des Java-Frameworks "JUnit 4" erstellt. Das Framework verwendet Annotationen, um sogenannte "Testsuiten" zu erstellen. Eine Suite wird durch eine Java-Klasse repräsentiert. In jeder Suite können mehrere Tests enthalten sein [41, Kapitel 3]. Die "Unit-Tests" für den "SyntaxTreeGenerator" sollen in solchen "Suiten" erstellt werden.

Die Vorgehensweise für "Unit-Tests" des "SyntaxTreeGenerators" basiert auf speziellen Dateien mit der Endung ".ast". Sie sind im Ordner "resources/syntaxtreegenerator" gespeichert und repräsentieren den AST der gleichnamige "Java TX"-Quelldatei aus dem Ordner "resources" (vgl. Anhang C auf Seite 77). Diese Dateien werden mit dem "ASTPrinter" erstellt,

der eine grafische Repräsentation von ASTs erstellen kann. Dazu nutzt er den sogenannten "ASTVisitor", der den AST eines Programms Knoten für Knoten durchläuft und entsprechende "Strings" erzeugt. Quelltext 3.13 zeigt dies am Beispiel des "SwitchLabels".

```
1    [...]
2    @Override
3    public void visit(SwitchLabel switchLabel) {
4      if (switchLabel.isDefault()) {
5        out.append("default");
6      } else {
7        out.append("case␣");
8        switchLabel.getExpression().accept(this);
9      }
10     out.append(":\n");
11   }
12   [...]
```

Quelltext 3.13: Konvertierung eines "SwitchLabel"-Knotens in Text

Innerhalb eines Tests wird die ".ast"-Datei per "FileInputStream" eingelesen. Anschließend wird eine Instanz des "JavaTXCompilers" erstellt, der die ".jav"-Datei des Tests einliest. Der erzeugte AST des Compilers wird dann mithilfe des "ASTVisitors" in seine grafische Form umgewandelt und mit dem Inhalt der ".ast"-Datei verglichen. Vorher werden jedoch noch die Bezeichner der "TypePlaceholder" aus den beiden "Strings" entfernt, da deren Werte nicht festgelegt sind. Für die Tests ist zudem nur relevant, ob sie die "TypePlaceholder" in beiden ASTs an derselben Stelle befinden.

Zum Vergleich des erwarteten mit dem generierten AST kommt die Funktion `assertEquals(String message, Object expected, Object actual)`, aus dem Framework "JUnit 4", zum Einsatz. Stimmen die ASTs, bzw. die Objekte "expected" und "actual", überein, ist der Test bestanden. Ansonsten wird eine Fehlermeldung generiert und der Test gilt als nicht bestanden [6, Abschnitt 3]. Quelltext 3.14 zeigt einen Test für den "SyntaxTreeGenerator" aus der Testsuite "TestComplete" im Paket "syntaxtreegenerator" (siehe Projektstruktur in Anhang C auf Seite 77). Die zugehörigen Dateien "Lambda.jav" und "Lambda.ast" sind in Anhang F.1 auf Seite 96 dargestellt.

```
1  [...]
2
```

```
3   @Test
4   public void lambdaTest() {
5     try {
6       FileInputStream fileIn = new FileInputStream(javFiles.get("Lambda
            ↪ ")[1]);
7       String expectedAST = new String(fileIn.readAllBytes());
8       fileIn.close();
9       expectedAST = expectedAST.replaceAll("TPH␣[A-Z]+", "TPH");
10      File srcfile = javFiles.get("Lambda")[0];
11      JavaTXCompiler compiler = new JavaTXCompiler(srcfile);
12      String resultingAST = new String(ASTPrinter.print(compiler.
            ↪ sourceFiles.get(srcfile)));
13      resultingAST = resultingAST.replaceAll("TPH␣[A-Z]+", "TPH");
14      System.out.println("Expected:\n" + new String(expectedAST));
15      System.out.println("Result:\n" + new String(resultingAST));
16      assertEquals("Comparing␣expected␣and␣resulting␣AST␣for␣Lambda.jav
            ↪ ", expectedAST, resultingAST);
17    } catch (Exception exc) {
18      exc.printStackTrace();
19      fail("An␣error␣occured␣while␣generating␣the␣AST␣for␣Lambda.jav");
20    }
21  }
22
23  [...]
```

Quelltext 3.14: Beispiel für einen "Unit-Test" des "SyntaxTreeGenerators"

Die Quelldatei wird auch in der gleichnamigen Testsuite für die Integrationstests des "Java TX"-Compilers verwendet und enthält keine neuen Programmstrukturen. Diese Tests wurden erstellt, um sicherzustellen, dass der "Java TX"-Compiler nach der Implementierung der neuen Funktionen wie gewohnt funktioniert. Quelldateien mit neuen Funktionen wie "switch"-Konstrukten oder "Records" werden in der separaten Testsuite "TestNewFeatures" getestet. Der Ablauf der Tests ist aber identisch. Ein Beispiel für eine "Java TX"-Quelldatei für die Suite "TestNewFeatures" ist in F.2 auf Seite 97 angehängt.

4 Zusammenfassung

Das folgenden Kapitel enthält Ausdrücke eigener Meinung und soll keine wissenschaftlichen Ansprüche erfüllen

Die ersten Schritte zur Implementierung aktueller Java-Funktionen in "Java TX" konnten in dieser Arbeit getätigt werden. Der AST des "Java TX"-Compilers unterstützt nun neue Sprachkonstrukte wie "Switch Expressions", "Records", "Sealed Classes" und "Pattern". Es wurden sowohl neue Klassen für die Repräsentation solcher Konstrukte im AST erstellt, als auch Methoden zum "SyntaxTreeGenerator", der für die Umwandlung von "Parsetrees" zu ASTs verantwortlich ist, hinzugefügt. Außerdem wurden "Unit-Tests" erstellt, um die korrekte Funktionsweise des "SyntaxTreeGenerators" zu prüfen. So kann sichergestellt werden, dass zukünftige Änderungen am Quellcode des "Java TX"-Compilers die Komponente nicht negativ beeinflussen.

Die Arbeit am "Java TX"-Compiler half mir bei der Verbesserung meiner Programmierfähigkeiten in objektorientierten Sprachen und schärfte mein Verständnis für die Funktionsweise von Compilern. Dazu trug vor allem die Erarbeitung der theoretischen Grundlagen in Kapitel 2 auf Seite 4 bei. Außerdem erhielt ich Einblick in die Forschungsarbeit an wissenschaftlichen Instituten und durfte erfahren, wie die Arbeit an Forschungsschwerpunkten abläuft.

4.1 Fazit

Leider konnten die neuen Funktionen, die "Pattern Matching" in "Java TX" ermöglichen sollen, nicht vollständig in den Compiler integriert werden. Die in Kapitel 1.3 auf Seite 3 formulierten Ziele konnten also nur teilweise erreicht werden. Mit der Anpassung des AST und des "SyntaxTreeGenerators" ist zwar ein wichtiger Schritt unternommen worden, die eigentlichen Kernkomponenten des "Java TX"-Compilers, der Typinferenzalgorithmus und der "Bytecode Genrator" wurden aber nicht erweitert. Die Anpassung der Komponenten für die semantische Analyse und das "Backend" des Compilers stehen also noch aus (vgl. Kapitel 2.5.4 auf Seite 33).

Die Anpassung aller Komponenten konnte innerhalb des Bearbeitungszeitraumes nicht erreicht werden. Dies hing, nach meiner Ansicht, vor allem mit fehlenden theoretischen Kenntnissen im Bereich Compilerbau und theoretischer Informatik zusammen. Dies führte dazu, dass einige Zeit aufgewendet werden musste, um den Compiler und zugrundeliegende Konzepte zu verstehen, bevor die Neuerungen tatsächlich implementiert werden konnten. Außerdem führten die Suche von Fehlern nach Implementierungsschritten und "Merges" der Änderungen anderer Entwickler mit dem Tool "git" (siehe Kapitel 3.1.1 auf Seite 49) zur Verzögerungen bei der Implementierung. Auch Probleme mit der Umstellung von Java Version 19 auf Java Version 20, und der damit zusammenhängenden Projektkonfiguration, verhinderten zeitweise einen Entwicklungsfortschritt. Es kam zum Beispiel zeitweise zur Problemen bei der Kompilierung des Compilers, da die "Maven"-Konfiguration inkompatible Versionseinträge enthielt.

Zusätzlich besteht für die Implementierungen aus Kapitel 3 auf Seite 48 mit hoher Wahrscheinlichkeit noch Optimierungspotenzial. Man könnte zum Beispiel die Klassen für Knoten im AST als "Records" implementieren. Die Verwendung von "Records" bietet sich an, da Instanzen der AST-Knoten genau solche Datenstrukturen repräsentieren, die mit "Records" optimalerweise implementiert werden sollten. Es handelt sich dabei um reine "Datenspeicher", die ein "Java TX"-Programm darstellen und nach ihrer Erstellung nicht mehr verändert werden.

Mit der Erweiterung des AST, des "SyntaxTreeGenerators" und der Erstellung der zugehörigen "Unit-Tests" konnten jedoch, trotz verfehlter Ziele, erste Schritte für die Aktualisierung von "Java TX" unternommen werden. Die Basis für eine Integration der neuen Funktionen rund um das "Pattern Matching" wurde gelegt.

4.2 Ausblick

Aufgrund der Tatsache, dass die neuen Sprachkonstrukte und Funktionen noch nicht vollständig vom "Java TX"-Compiler verarbeitet werden können, verbleiben einige Tätigkeiten, um die Integration der neuen Funktionen abzuschließen. Außerdem besteht die Möglichkeit, "Java TX", unabhängig von der Sprache Java, weiter an funktionale Programmiersprachen anzunähern.

4.2.1 Verbleibende Tätigkeiten

Bevor der "Java TX"-Compiler neue Konstrukte und Funktionen wie "Switch" oder den "instanceof"-Operator in JVM-Bytecode übersetzen kann, sind weitere Implementierungsschritte notwendig. Zunächst muss der Typinferenzalgorithmus angepasst werden. Dabei müssen zunächst "Constraints" für die Typen und "TypePlaceholder" erzeugt werden, die in den neu implementierten Sprachstrukturen vorkommen (siehe "type"-Felder in den Klassen aus Kapitel 3.2 auf Seite 50). Diese "Constraints" müssen dann während der "Typunifikation" berücksichtigt werden, um sämtliche Typen eines "neuen Java TX Programmes" inferieren zu können.

Sobald der "Typinferenzalgorithmus" mit "Pattern", "Switch"-Konstrukten, etc. umgehen kann, ist der Weg frei für den letzten Anpassungsschritt. Der "Bytecode Generator" (Klasse "Codegen") kann nun erweitert werden, so dass er, mit dem AST und den Ergebnissen aus dem "ResultSet" der Typinferenz, den passenden JVM-Bytecode für ein "Java TX"-Programm erzeugen kann (vgl. 2.1 auf Seite 6). Abschließend sollten die Quelldateien, die aktuell für die "Unit-Tests" des "SyntaxTreeGenerators" in "TestNewFeatures" verwendet werden, auch in die Testsuite für Integrationstests ("TestComplete") aufgenommen werden. So kann der vollständig aktualisierte Compiler auf Fehler geprüft werden.

4.2.2 "Haskell Pattern Matching"

Zur Weiterentwicklung von "Java TX" nach dem Abschluss der ausstehenden Entwicklungsarbeiten könnten zusätzliche Funktionen funktionaler Programmiersprachen als Vorlage dienen. So bietet die Sprache "Haskell" beispielsweise die Möglichkeit, Methoden mithilfe von "Pattern Matching" zu überladen. Je nach "Pattern", in der Parameterliste der Funktion bzw. Methode, unterscheidet sich das Verhalten der Methode. Dieses Feature würde den Entwicklern zusätzliche Flexibilität bei ihrer Arbeit mit "Java TX" bieten und könnte für zusätzliche Effizienz sorgen. Beispiele für die Anwendung dieser Art von "Pattern Matching" zeigt []

A Parsebaum für “Java TX”-Quelldatei

A.1 “Java TX”-Quelldatei “applyLambda.jav”

```
1  import java.util.Vector;
2  class Apply { }
3
4  public class applyLambda {
5
6    m () {
7     var lam1 = (x) -> {
8         return x;
9     };
10
11    return lam1.apply(new Apply());
12    //return lam1;
13    //return new Vector();
14   }
15 }
```

Quelltext A.1: “Java TX”-Quelldatei “applyLambda.jav”

A.2 Ableitungsbaum als Ergebnis der sytaktischen Analyse von "applyLambda.jav"

```
1  sourceFile
2    importDeclaration import
3      qualifiedName
4        identifier java
5        .
6        identifier util
7        .
8        identifier Vector
9      ;
10   classOrInterface
11     classDeclaration class
12       identifier Apply
13       classBody {}
14   classOrInterface
15     classOrInterfaceModifier public
16     classDeclaration class
17       identifier applyLambda
18       classBody {
19         classBodyDeclaration
20           memberDeclaration
21             method
22               methodDeclaration
23                 methodHeader
24                   identifier m
25                   formalParameters ()
26                 methodBody
27                   block {
28                     blockStatement
29                       localVariableDeclaration var
30                         variableDeclarators
31                           variableDeclarator
32                             variableDeclaratorId
```

```
33                    identifier lam1
34                   =
35                    variableInitializer
36                      expression
37                        lambdaExpression
38                          lambdaParameters (
39                            formalParameterList
40                              formalParameter
41                                variableDeclaratorId
42                                  identifier x
43                            )
44                          ->
45                          lambdaBody
46                            block {
47                              blockStatement
48                                statement return
49                                  expression
50                                    primary
51                                      identifier x
52                                  ;
53                              }
54              ;
55            blockStatement
56              statement return
57                expression
58                  expression
59                    primary
60                      identifier lam1
61                  .
62                  methodCall
63                    identifier apply
64                    (
65                    expressionList
66                      expression new
67                        creator
```

```
68                                  createdName
69                                    identifier Apply
70                                  classCreatorRest
71                                    arguments ()
72                              )
73                          ;
74                      }
75          }
```

Quelltext A.2: Ableitungsbaum als Ergebnis der sytaktischen Analyse von "applyLambda.jav"

B Bytecode der Klasse "PrintNames"

Die Klasse "PrintNames" enthält zwei Methoden. "printNamesWithSugar" (siehe Quelltext 2.8 auf Seite 20) und "printNames" (siehe Quelltext 2.9 auf Seite 21). Die Gegenüberstellung des Bytecodes der beiden Methoden verdeutlicht die geringen Auswirkungen von syntaktische Veränderungen durch "Syntactic Sugar" auf den generierten JVM Bytecode.

```
1  void printNamesWithSugar();
2  Code:
3     0: iconst_4
4     1: anewarray #7
5     4: dup
6     5: iconst_0
7     6: ldc #9
8     8: aastore
9     9: dup
10   10: iconst_1
11   11: ldc #11
12   13: aastore
13   14: dup
14   15: iconst_2
15   16: ldc #13
16   18: aastore
17   19: dup
18   20: iconst_3
19   21: ldc #15
20   23: aastore
21   24: invokestatic #17
22   27: astore_1
23   28: aload_1
24   29: invokeinterface #23, 1
25   34: astore_2
26   35: aload_2
27   36: invokeinterface #29, 1
28   41: ifeq 64
29   44: aload_2
30   45: invokeinterface #35, 1
31   50: checkcast #7
32   53: astore_3
33   54: getstatic #39
34   57: aload_3
35   58: invokevirtual #45
36   61: goto 35
37   64: return
```

Quelltext B.1: Bytecode für die Methode "printNamesWithSugar"

```
1   void printNames();
2   Code:
3      0: iconst_4
4      1: anewarray #7
5      4: dup
6      5: iconst_0
7      6: ldc #9
8      8: aastore
9      9: dup
10    10: iconst_1
11    11: ldc #11
12    13: aastore
13    14: dup
14    15: iconst_2
15    16: ldc #13
16    18: aastore
17    19: dup
18    20: iconst_3
19    21: ldc #15
20    23: aastore
21    24: invokestatic #17
22    27: astore_1
23    28: iconst_0
24    29: istore_2
25    30: iload_2
26    31: aload_1
27    32: invokeinterface #51, 1
28    37: if_icmpge 62
29    40: getstatic #39
30    43: aload_1
31    44: iload_2
32    45: invokeinterface #55, 2
33    50: checkcast #7
34    53: invokevirtual #45
35    56: iinc 2, 1
36    59: goto 30
37    62: return
```

Quelltext B.2: Bytecode für die Methode "printNames"

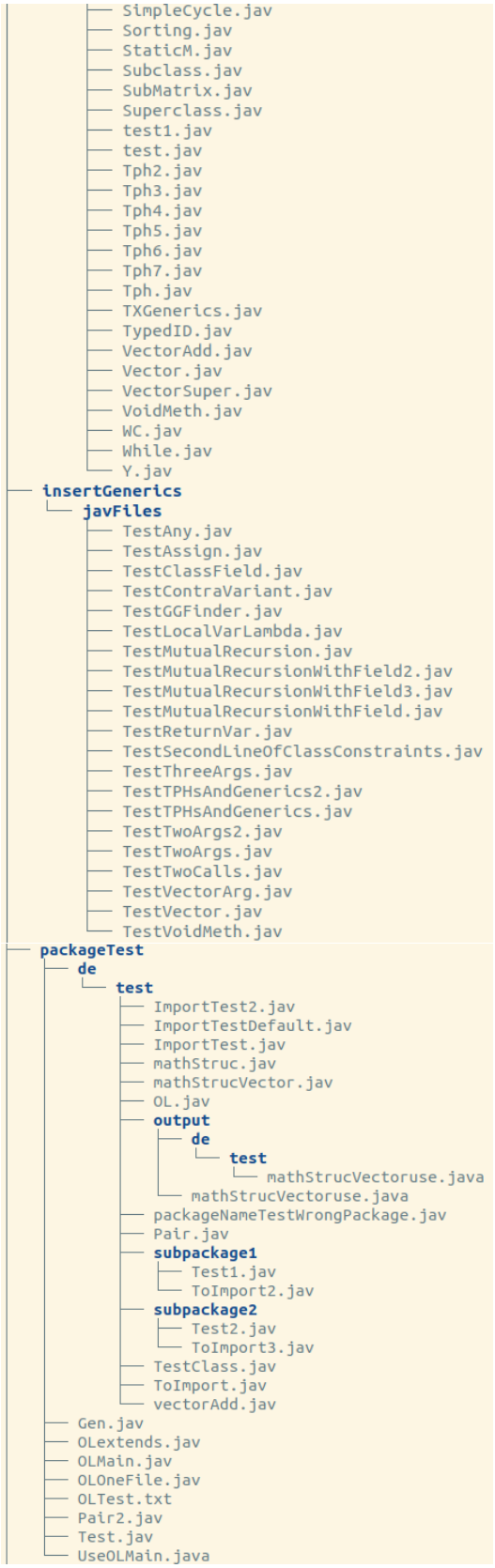
C Projektstruktur

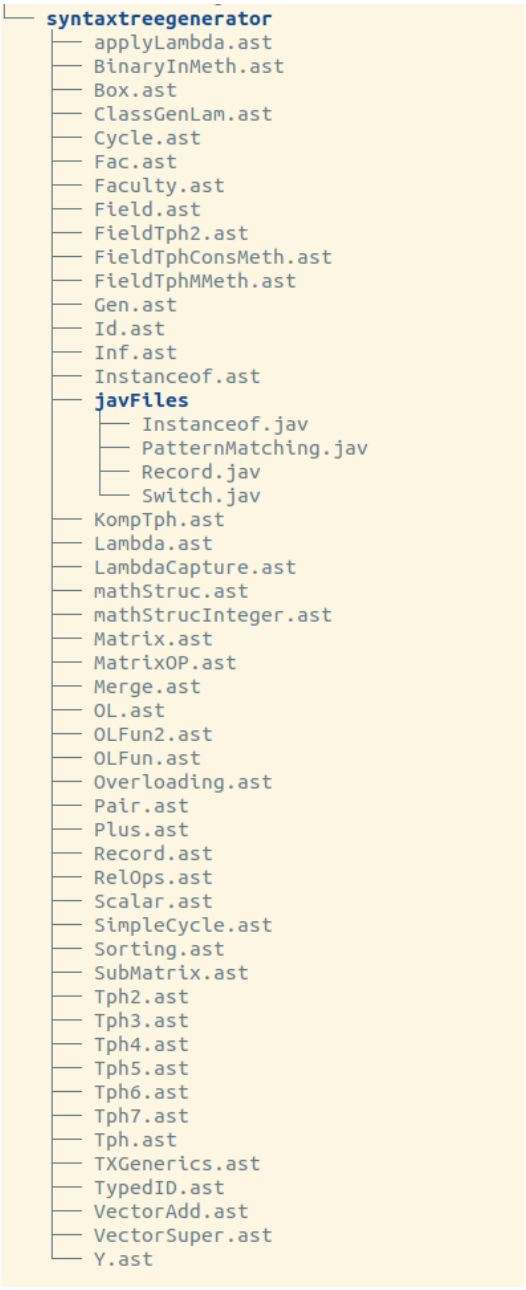
C.1 Übersicht über die Projektstruktur des "Java TX"-Compilers

Vollständige Übersicht der Verzeichnisse "resources" und "src" aus dem Hauptverzeichnis des "JavaCompilerCore"-Projekts. Auf die Darstellung der Binärdateien mit der Endung ".class" sowie dem Ordner "target" wurde für die Wahrung einer gewissen Übersichtlichkeit verzichtet.

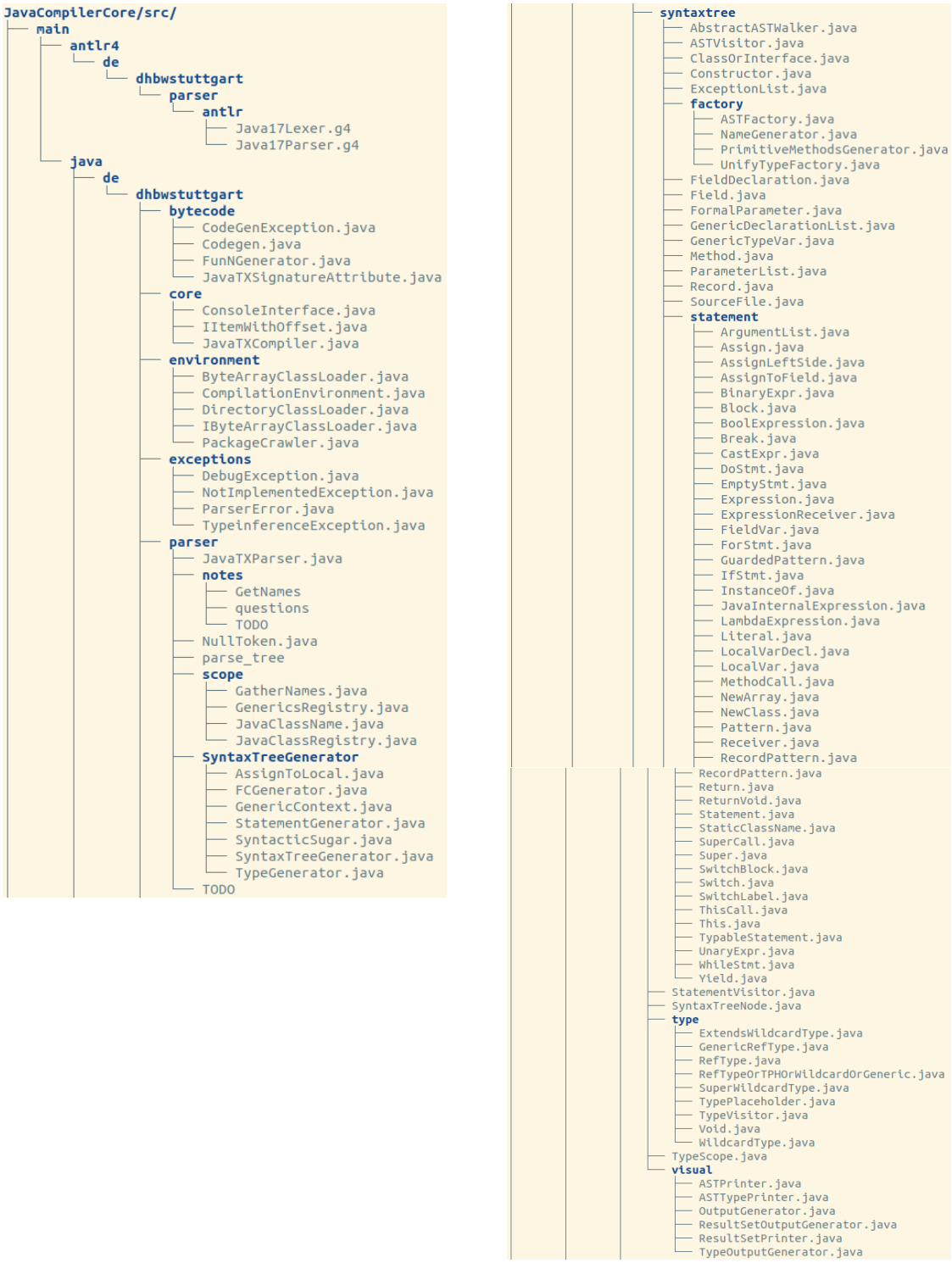
C.1.1 Verzeichnis "JavaCompilerCore/resources"

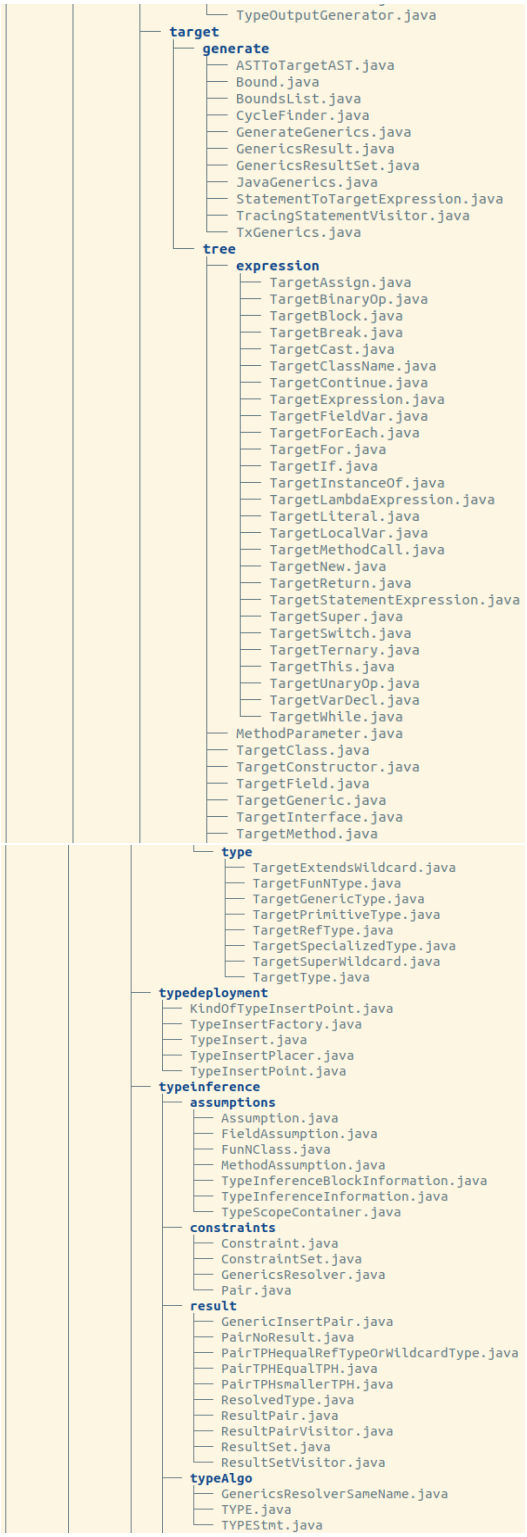
```
JavaCompilerCore/resources/
├── AllgemeinTest
│   ├── Assign.jav
│   ├── CaptureConversion.jav
│   ├── Iteration.jav
│   ├── Pair.jav
│   ├── StreamTest.jav
│   └── UseWildcardPair.jav
├── bytecode
│   └── javFiles
│       ├── AA.jav
│       ├── AddLong.jav
│       ├── applyLambda.jav
│       ├── AssignToLit.jav
│       ├── BB.jav
│       ├── BinaryInMeth.jav
│       ├── Box.jav
│       ├── Box.java
│       ├── CC.jav
│       ├── ClassGenLam.jav
│       ├── Cycle.class
│       ├── Cycle.jav
│       ├── DD.jav
│       ├── DuMethod.jav
│       ├── EmptyClass.jav
│       ├── EmptyMethod.jav
│       ├── Example.jav
│       ├── Exceptions.jav
│       ├── Expressions.jav
│       ├── Fac.jav
│       ├── Faculty2.jav
│       ├── FacultyIf.jav
│       ├── Faculty.jav
│       ├── FacultyTyped.jav
│       ├── fc.jav
│       ├── FC_Matrix.jav
│       ├── FieldAccess.jav
│       ├── Field.jav
│       ├── Fields.jav
│       ├── FieldTph2.jav
│       ├── FieldTphConsMeth.jav
│       ├── FieldTph.jav
│       ├── FieldTphMMeth.jav
│       ├── For.jav
│       ├── FunOL.jav
│       ├── Generics2.jav
│       ├── Generics3.jav
│       ├── Generics4.jav
│       ├── Generics.jav
│       ├── Gen.jav
│       ├── GreaterEqual.jav
│       ├── GreaterThan.jav
│       ├── Id.jav
│       ├── IfTest.jav
│       ├── Import.jav
│       ├── Infimum.jav
│       ├── Inf.jav
│       ├── Inherit2.jav
│       ├── Inherit.jav
│       ├── Interface1.jav
│       ├── KompTph.jav
│       ├── Lambda2.jav
│       ├── Lambda3.jav
│       ├── Lambda4.jav
│       ├── LambdaCapture.jav
│       ├── LambdaField.jav
│       ├── Lambda.jav
│       ├── LambdaRunnable.jav
│       ├── LambdaVoid.jav
│       ├── LamRunnable.jav
│       ├── LessEqual.jav
│       ├── LessThan.jav
│       ├── ListenerOverload.jav
│       ├── mathStrucInteger.jav
│       ├── mathStruc.jav
│       ├── mathStrucMatrixOP.jav
│       ├── Matrix.jav
│       ├── MatrixOP.jav
│       ├── Merge.jav
│       ├── Meth_Gen.jav
│       ├── MethodCallGenerics.jav
│       ├── MethodsEasy.jav
│       ├── Methods.jav
│       ├── MethodWildcardGen.jav
│       ├── OLFun2.jav
│       ├── OLFun.jav
│       ├── OL.jav
│       ├── Op1.jav
│       ├── Op2.jav
│       ├── Op.jav
│       ├── OverlaodGen.jav
│       ├── Overloading.jav
│       ├── Package.jav
│       ├── Pair.jav
│       ├── Plus.jav
│       ├── PostIncDec.jav
│       ├── PreInc.jav
│       ├── Put.jav
│       ├── RecursiveMeth.jav
│       ├── RelOps.jav
│       ├── ReturnMethod.jav
│       ├── Scalar.jav
│       ├── SimpleCycle.jav
```

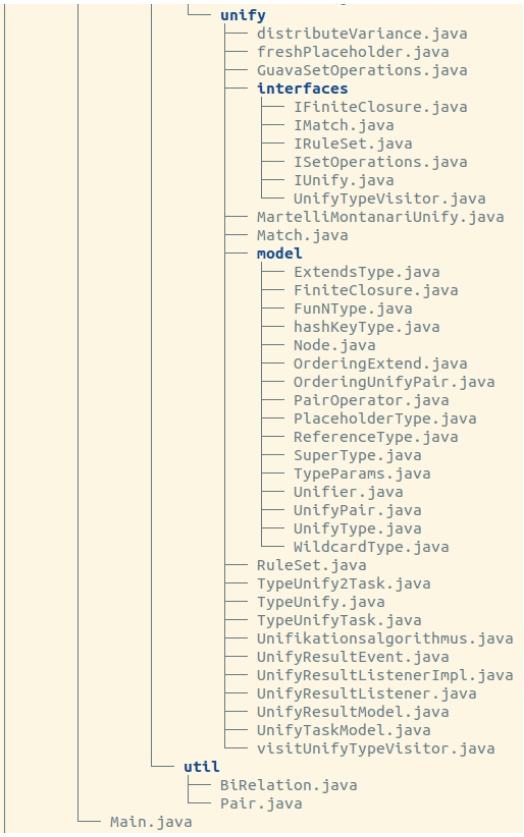

SimpleCycle.jav
Sorting.jav
StaticM.jav
Subclass.jav
SubMatrix.jav
Superclass.jav
test1.jav
test.jav
Tph2.jav
Tph3.jav
Tph4.jav
Tph5.jav
Tph6.jav
Tph7.jav
Tph.jav
TXGenerics.jav
TypedID.jav
VectorAdd.jav
Vector.jav
VectorSuper.jav
VoidMeth.jav
WC.jav
While.jav
Y.jav
insertGenerics
javFiles
TestAny.jav
TestAssign.jav
TestClassField.jav
TestContraVariant.jav
TestGGFinder.jav
TestLocalVarLambda.jav
TestMutualRecursion.jav
TestMutualRecursionWithField2.jav
TestMutualRecursionWithField3.jav
TestMutualRecursionWithField.jav
TestReturnVar.jav
TestSecondLineOfClassConstraints.jav
TestThreeArgs.jav
TestTPHsAndGenerics2.jav
TestTPHsAndGenerics.jav
TestTwoArgs2.jav
TestTwoArgs.jav
TestTwoCalls.jav
TestVectorArg.jav
TestVector.jav
TestVoidMeth.jav
packageTest
de
test
ImportTest2.jav
ImportTestDefault.jav
ImportTest.jav
mathStruc.jav
mathStrucVector.jav
OL.jav
output
de
test
mathStrucVectoruse.java
mathStrucVectoruse.java
packageNameTestWrongPackage.jav
Pair.jav
subpackage1
Test1.jav
ToImport2.jav
subpackage2
Test2.jav
ToImport3.jav
TestClass.jav
ToImport.jav
vectorAdd.jav
Gen.jav
OLextends.jav
OLMain.jav
OLOneFile.jav
OLTest.txt
Pair2.jav
Test.jav
UseOLMain.java

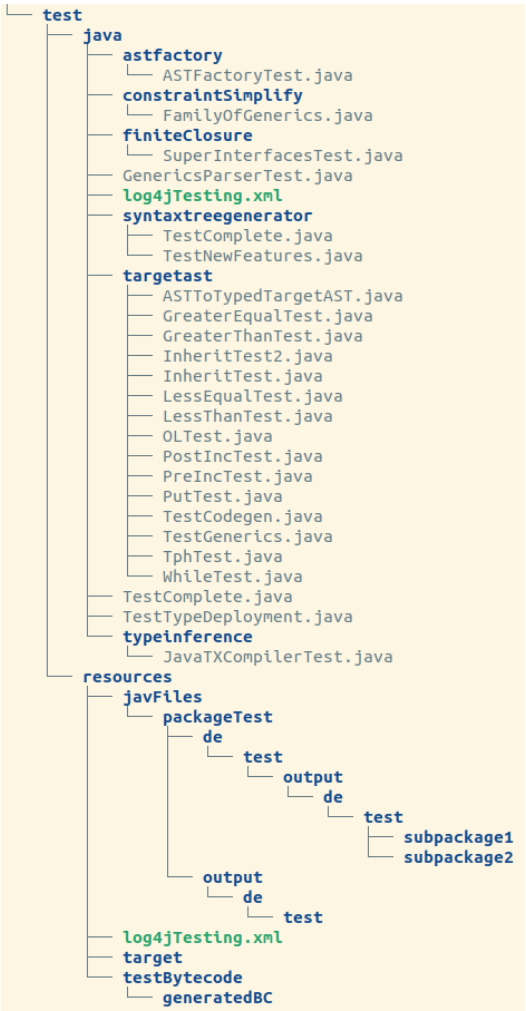

syntaxtreegenerator
applyLambda.ast
BinaryInMeth.ast
Box.ast
ClassGenLam.ast
Cycle.ast
Fac.ast
Faculty.ast
Field.ast
FieldTph2.ast
FieldTphConsMeth.ast
FieldTphMMeth.ast
Gen.ast
Id.ast
Inf.ast
Instanceof.ast
javFiles
Instanceof.jav
PatternMatching.jav
Record.jav
Switch.jav
KompTph.ast
Lambda.ast
LambdaCapture.ast
mathStruc.ast
mathStrucInteger.ast
Matrix.ast
MatrixOP.ast
Merge.ast
OL.ast
OLFun2.ast
OLFun.ast
Overloading.ast
Pair.ast
Plus.ast
Record.ast
RelOps.ast
Scalar.ast
SimpleCycle.ast
Sorting.ast
SubMatrix.ast
Tph2.ast
Tph3.ast
Tph4.ast
Tph5.ast
Tph6.ast
Tph7.ast
Tph.ast
TXGenerics.ast
TypedID.ast
VectorAdd.ast
VectorSuper.ast
Y.ast

C.1.2 Verzeichnis “JavaCompilerCore/src”




TypeOutputGenerator.java
target
generate
ASTToTargetAST.java
Bound.java
BoundsList.java
CycleFinder.java
GenerateGenerics.java
GenericsResult.java
GenericsResultSet.java
JavaGenerics.java
StatementToTargetExpression.java
TracingStatementVisitor.java
TxGenerics.java
tree
expression
TargetAssign.java
TargetBinaryOp.java
TargetBlock.java
TargetBreak.java
TargetCast.java
TargetClassName.java
TargetContinue.java
TargetExpression.java
TargetFieldVar.java
TargetForEach.java
TargetFor.java
TargetIf.java
TargetInstanceOf.java
TargetLambdaExpression.java
TargetLiteral.java
TargetLocalVar.java
TargetMethodCall.java
TargetNew.java
TargetReturn.java
TargetStatementExpression.java
TargetSuper.java
TargetSwitch.java
TargetTernary.java
TargetThis.java
TargetUnaryOp.java
TargetVarDecl.java
TargetWhile.java
MethodParameter.java
TargetClass.java
TargetConstructor.java
TargetField.java
TargetGeneric.java
TargetInterface.java
TargetMethod.java
type
TargetExtendsWildcard.java
TargetFunNType.java
TargetGenericType.java
TargetPrimitiveType.java
TargetRefType.java
TargetSpecializedType.java
TargetSuperWildcard.java
TargetType.java
typedeployment
KindOfTypeInsertPoint.java
TypeInsertFactory.java
TypeInsert.java
TypeInsertPlacer.java
TypeInsertPoint.java
typeinference
assumptions
Assumption.java
FieldAssumption.java
FunNClass.java
MethodAssumption.java
TypeInferenceBlockInformation.java
TypeInferenceInformation.java
TypeScopeContainer.java
constraints
Constraint.java
ConstraintSet.java
GenericsResolver.java
Pair.java
result
GenericInsertPair.java
PairNoResult.java
PairTPHequalRefTypeOrWildcardType.java
PairTPHEqualTPH.java
PairTPHsmallerTPH.java
ResolvedType.java
ResultPair.java
ResultPairVisitor.java
ResultSet.java
ResultSetVisitor.java
typeAlgo
GenericsResolverSameName.java
TYPE.java
TYPEStmt.java


unify
distributeVariance.java
freshPlaceholder.java
GuavaSetOperations.java
interfaces
IFiniteClosure.java
IMatch.java
IRuleSet.java
ISetOperations.java
IUnify.java
UnifyTypeVisitor.java
MartelliMontanariUnify.java
Match.java
model
ExtendsType.java
FiniteClosure.java
FunNType.java
hashKeyType.java
Node.java
OrderingExtend.java
OrderingUnifyPair.java
PairOperator.java
PlaceholderType.java
ReferenceType.java
SuperType.java
TypeParams.java
Unifier.java
UnifyPair.java
UnifyType.java
WildcardType.java
RuleSet.java
TypeUnify2Task.java
TypeUnify.java
TypeUnifyTask.java
Unifikationsalgorithmus.java
UnifyResultEvent.java
UnifyResultListenerImpl.java
UnifyResultListener.java
UnifyResultModel.java
UnifyTaskModel.java
visitUnifyTypeVisitor.java
util
BiRelation.java
Pair.java
Main.java


test
java
astfactory
ASTFactoryTest.java
constraintSimplify
FamilyOfGenerics.java
finiteClosure
SuperInterfacesTest.java
GenericsParserTest.java
log4jTesting.xml
syntaxtreegenerator
TestComplete.java
TestNewFeatures.java
targetast
ASTToTypedTargetAST.java
GreaterEqualTest.java
GreaterThanTest.java
InheritTest2.java
InheritTest.java
LessEqualTest.java
LessThanTest.java
OLTest.java
PostIncTest.java
PreIncTest.java
PutTest.java
TestCodegen.java
TestGenerics.java
TphTest.java
WhileTest.java
TestComplete.java
TestTypeDeployment.java
typeinference
JavaTXCompilerTest.java
resources
javFiles
packageTest
de
test
output
de
test
subpackage1
subpackage2
output
de
test
log4jTesting.xml
target
testBytecode
generatedBC

D "Java TX"-Bytecode

Der dargestellte Bytecode ermöglicht die Ausführung des Programms 2.12 auf Seite 29 in einer JVM.

```
1  public class Fac {
2    public Fac();
3      Code:
4         0: aload_0
5         1: invokespecial #9 // Method java/lang/Object."<init>":()V
6         4: return
7
8    public java.lang.Double getFac(java.lang.Double);
9      Code:
10        0: aconst_null
11        1: astore_2
12        2: ldc #12 // int 1
13        4: i2d
14        5: invokestatic #18 // Method java/lang/Double.valueOf:(D)Ljava/
             ↪ lang/Double;
15        8: dup
16        9: astore_2
17       10: pop
18       11: aconst_null
19       12: astore_3
20       13: ldc #12 // int 1
21       15: i2d
22       16: invokestatic #18 // Method java/lang/Double.valueOf:(D)Ljava/
            ↪ lang/Double;
23       19: dup
24       20: astore_3
25       21: pop
26       22: aload_3
27       23: invokevirtual #22 // Method java/lang/Double.doubleValue:()D
```

```
28        26: aload_1
29        27: invokevirtual #22 // Method java/lang/Double.doubleValue:()D
30        30: dcmpl
31        31: ifle 38
32        34: iconst_0
33        35: goto 39
34        38: iconst_1
35        39: ifeq 74
36        42: aload_2
37        43: invokevirtual #22 // Method java/lang/Double.doubleValue:()D
38        46: aload_3
39        47: invokevirtual #22 // Method java/lang/Double.doubleValue:()D
40        50: dmul
41        51: invokestatic #18 // Method java/lang/Double.valueOf:(D)Ljava/
              ↪ lang/Double;
42        54: dup
43        55: astore_2
44        56: pop
45        57: aload_3
46        58: invokevirtual #22 // Method java/lang/Double.doubleValue:()D
47        61: dup2
48        62: ldc2_w #23 // double 1.0d
49        65: dadd
50        66: invokestatic #18 // Method java/lang/Double.valueOf:(D)Ljava/
              ↪ lang/Double;
51        69: astore_3
52        70: pop2
53        71: goto 22
54        74: aload_2
55        75: invokevirtual #22 // Method java/lang/Double.doubleValue:()D
56        78: invokestatic #18 // Method java/lang/Double.valueOf:(D)Ljava/
              ↪ lang/Double;
57        81: areturn
58
59    public java.lang.Integer getFac(java.lang.Integer);
```

```
60      Code:
61         0: aconst_null
62         1: astore_2
63         2: ldc #12 // int 1
64         4: invokestatic #30 // Method java/lang/Integer.valueOf:(I)Ljava
              ↪ /lang/Integer;
65         7: dup
66         8: astore_2
67         9: pop
68        10: aconst_null
69        11: astore_3
70        12: ldc #12 // int 1
71        14: invokestatic #30 // Method java/lang/Integer.valueOf:(I)Ljava
             ↪ /lang/Integer;
72        17: dup
73        18: astore_3
74        19: pop
75        20: aload_3
76        21: invokevirtual #34 // Method java/lang/Integer.intValue:()I
77        24: aload_1
78        25: invokevirtual #34 // Method java/lang/Integer.intValue:()I
79        28: if_icmple 35
80        31: iconst_0
81        32: goto 36
82        35: iconst_1
83        36: ifeq 70
84        39: aload_2
85        40: invokevirtual #34 // Method java/lang/Integer.intValue:()I
86        43: aload_3
87        44: invokevirtual #34 // Method java/lang/Integer.intValue:()I
88        47: imul
89        48: invokestatic #30 // Method java/lang/Integer.valueOf:(I)Ljava
             ↪ /lang/Integer;
90        51: dup
91        52: astore_2
```

```
92        53: pop
93        54: aload_3
94        55: invokevirtual #34 // Method java/lang/Integer.intValue:()I
95        58: dup
96        59: ldc #12 // int 1
97        61: iadd
98        62: invokestatic #30 // Method java/lang/Integer.valueOf:(I)Ljava
              ↪ /lang/Integer;
99        65: astore_3
100       66: pop
101       67: goto 20
102       70: aload_2
103       71: invokevirtual #34 // Method java/lang/Integer.intValue:()I
104       74: invokestatic #30 // Method java/lang/Integer.valueOf:(I)Ljava
              ↪ /lang/Integer;
105       77: areturn
106 }
```

Quelltext D.1: Bytecode für Programm "Fac.jav"

E Neue Knoten im abstrakten Syntaxbaum von “Java TX”

E.1 “Record”

```
1  package de.dhbwstuttgart.syntaxtree;
2
3  import java.util.ArrayList;
4  import java.util.List;
5  import java.util.Optional;
6
7  import org.antlr.v4.runtime.Token;
8
9  import de.dhbwstuttgart.parser.scope.JavaClassName;
10 import de.dhbwstuttgart.syntaxtree.type.RefType;
11
12 public class Record extends ClassOrInterface {
13
14   public Record(int modifiers, JavaClassName name, List<Field>
       ↪ fielddecl, Optional<Constructor> fieldInitializations, List<
       ↪ Method> methods, List<Constructor> constructors,
       ↪ GenericDeclarationList genericClassParameters, RefType
       ↪ superClass, Boolean isInterface, List<RefType>
       ↪ implementedInterfaces, Token offset) {
15     super(modifiers, name, fielddecl, fieldInitializations, methods,
         ↪ constructors, genericClassParameters, superClass, isInterface
         ↪ , implementedInterfaces, new ArrayList<>(), offset);
16   }
17 }
```

Quelltext E.1: Klasse “Record” zur Verwendung im AST von “Java TX”

E.2 "Instanceof"

```
1  package de.dhbwstuttgart.syntaxtree.statement;
2
3  import org.antlr.v4.runtime.Token;
4
5  import de.dhbwstuttgart.syntaxtree.StatementVisitor;
6  import de.dhbwstuttgart.syntaxtree.type.
       ↪ RefTypeOrTPHOrWildcardOrGeneric;
7  import de.dhbwstuttgart.syntaxtree.type.TypePlaceholder;
8
9  public class InstanceOf extends BinaryExpr {
10     private RefTypeOrTPHOrWildcardOrGeneric reftype;
11     private String name = null;
12
13     public InstanceOf(Expression expr, RefTypeOrTPHOrWildcardOrGeneric
           ↪ reftype, Token offset) {
14         super(BinaryExpr.Operator.INSTOF, TypePlaceholder.fresh(offset)
               ↪ , expr, new LocalVar("", reftype, reftype.getOffset()),
               ↪ offset);
15         this.reftype = reftype;
16     }
17
18     public InstanceOf(Expression expr, Pattern pattern, Token offset) {
19         super(BinaryExpr.Operator.INSTOF, TypePlaceholder.fresh(offset)
               ↪ , expr, new LocalVar(pattern.getName(), pattern.getType()
               ↪ , pattern.getOffset()), offset);
20         this.reftype = pattern.getType();
21         this.name = pattern.getName();
22     }
23
24     public RefTypeOrTPHOrWildcardOrGeneric getReftype() {
25         return reftype;
26     }
27
```

```
28      public String getName() {
29          return name;
30      }
31
32      @Override
33      public void accept(StatementVisitor visitor) {
34          visitor.visit(this);
35      }
36  }
```

Quelltext E.2: Klasse "Instanceof" zur Verwendung im AST von "Java TX"

E.3 "Pattern"

```
1   package de.dhbwstuttgart.syntaxtree.statement;
2
3   import org.antlr.v4.runtime.Token;
4
5   import de.dhbwstuttgart.syntaxtree.StatementVisitor;
6   import de.dhbwstuttgart.syntaxtree.type.
        ↪ RefTypeOrTPHOrWildcardOrGeneric;
7
8   public class Pattern extends Expression {
9
10      private String name;
11
12      public Pattern(String name, RefTypeOrTPHOrWildcardOrGeneric type,
            ↪ Token offset) {
13          super(type, offset);
14          this.name = name;
15      }
16
17      public String getName() {
18          return name;
19      }
20
```

```
21      @Override
22      public void accept(StatementVisitor visitor) {
23          visitor.visit(this);
24      }
25
26  }
```

Quelltext E.3: Klasse "Pattern" zur Verwendung im AST von "Java TX"

E.4 "GuardedPattern"

```
1   package de.dhbwstuttgart.syntaxtree.statement;
2
3   import java.util.List;
4
5   import org.antlr.v4.runtime.Token;
6
7   import de.dhbwstuttgart.syntaxtree.type.
        ↪ RefTypeOrTPHOrWildcardOrGeneric;
8
9   public class GuardedPattern extends Pattern {
10
11      private List<Expression> conditions;
12
13      public GuardedPattern(List<Expression> conditions, String name,
            ↪ RefTypeOrTPHOrWildcardOrGeneric type, Token offset) {
14          super(name, type, offset);
15          this.conditions = conditions;
16      }
17
18      public List<Expression> getConditions() {
19          return conditions;
20      }
21
22  }
```

Quelltext E.4: Klasse "GuardedPattern" zur Verwendung im AST von "Java TX"

E.5 "RecordPattern"

```
1  package de.dhbwstuttgart.syntaxtree.statement;
2
3  import java.util.ArrayList;
4  import java.util.List;
5
6  import org.antlr.v4.runtime.Token;
7
8  import de.dhbwstuttgart.syntaxtree.StatementVisitor;
9  import de.dhbwstuttgart.syntaxtree.type.
       ↪ RefTypeOrTPHOrWildcardOrGeneric;
10
11 public class RecordPattern extends Pattern {
12
13   private List<Pattern> subPattern = new ArrayList<>();
14
15   public RecordPattern(String name, RefTypeOrTPHOrWildcardOrGeneric
         ↪ type, Token offset) {
16     super(name, type, offset);
17   }
18
19   public RecordPattern(List<Pattern> subPattern, String name,
         ↪ RefTypeOrTPHOrWildcardOrGeneric type, Token offset) {
20     super(name, type, offset);
21     this.subPattern = subPattern;
22   }
23
24   public List<Pattern> getSubPattern() {
25     return this.subPattern;
26   }
27
28   public void addSubPattern(Pattern newPattern) {
29     this.subPattern.add(newPattern);
30   }
```

```
31
32    @Override
33    public void accept(StatementVisitor visitor) {
34      visitor.visit(this);
35    }
36
37  }
```

Quelltext E.5: Klasse "RecordPattern" zur Verwendung im AST von "Java TX"

E.6 "Switch"

```
1  package de.dhbwstuttgart.syntaxtree.statement;
2
3  import java.util.ArrayList;
4  import java.util.List;
5
6  import org.antlr.v4.runtime.Token;
7
8  import de.dhbwstuttgart.syntaxtree.StatementVisitor;
9  import de.dhbwstuttgart.syntaxtree.type.
       ↪ RefTypeOrTPHOrWildcardOrGeneric;
10
11 public class Switch extends Statement {
12
13   private Expression switchedExpression;
14   private List<SwitchBlock> blocks = new ArrayList<>();
15
16   public Switch(Expression switched, List<SwitchBlock> blocks,
         ↪ RefTypeOrTPHOrWildcardOrGeneric type, Boolean isStatement,
         ↪ Token offset) {
17     super(type, offset);
18     if (isStatement)
19       setStatement();
20     this.switchedExpression = switched;
21     this.blocks = blocks;
```

```
22    }
23
24    public Expression getSwitch() {
25      return switchedExpression;
26    }
27
28    public List<SwitchBlock> getBlocks() {
29      return blocks;
30    }
31
32    @Override
33    public void accept(StatementVisitor visitor) {
34      visitor.visit(this);
35    }
36
37  }
```

Quelltext E.6: Klasse "Switch" zur Verwendung im AST von "Java TX"

E.7 "SwitchBlock"

```
1  package de.dhbwstuttgart.syntaxtree.statement;
2
3  import java.util.ArrayList;
4  import java.util.List;
5
6  import org.antlr.v4.runtime.Token;
7
8  import de.dhbwstuttgart.syntaxtree.StatementVisitor;
9
10 public class SwitchBlock extends Block {
11
12     private List<SwitchLabel> labels = new ArrayList<>();
13
14     private Boolean defaultBlock = false;
15
```

```
16      public SwitchBlock(List<SwitchLabel> labels, Block statements,
            ↪ Token offset) {
17          super(statements.getStatements(), offset);
18          this.labels = labels;
19      }
20
21      public SwitchBlock(List<SwitchLabel> labels, Block statements,
            ↪ Boolean isDefault, Token offset) {
22          super(statements.getStatements(), offset);
23          this.labels = labels;
24          this.defaultBlock = isDefault;
25      }
26
27      public Boolean isDefault() {
28          return defaultBlock;
29      }
30
31      public List<SwitchLabel> getLabels() {
32          return labels;
33      }
34
35      @Override
36      public void accept(StatementVisitor visitor) {
37          visitor.visit(this);
38      }
39
40  }
```

Quelltext E.7: Klasse "SwitchBlock" zur Verwendung im AST von "Java TX"

E.8 "SwitchLabel"

```
1  package de.dhbwstuttgart.syntaxtree.statement;
2
3  import org.antlr.v4.runtime.Token;
4
5  import de.dhbwstuttgart.syntaxtree.StatementVisitor;
6  import de.dhbwstuttgart.syntaxtree.type.
       ↪ RefTypeOrTPHOrWildcardOrGeneric;
7
8  public class SwitchLabel extends Expression {
9
10     private Expression caseExpression;
11     private Boolean defaultCase = false;
12
13     public SwitchLabel(Expression caseExpression,
           ↪ RefTypeOrTPHOrWildcardOrGeneric type, Token offset) {
14         super(type, offset);
15         this.caseExpression = caseExpression;
16     }
17
18     public SwitchLabel(RefTypeOrTPHOrWildcardOrGeneric type, Token
           ↪ offset) {
19         super(type, offset);
20         this.defaultCase = true;
21     }
22
23     public Expression getExpression() {
24         return caseExpression;
25     }
26
27     public Boolean isDefault() {
28         return this.defaultCase;
29     }
30
```

```
31      @Override
32      public void accept(StatementVisitor visitor) {
33          visitor.visit(this);
34      }
35
36  }
```

Quelltext E.8: Klasse "SwitchLabel" zur Verwendung im AST von "Java TX"

F Dateien für “SyntaxTreeGenerator”-Tests

F.1 Quelltext und grafischer AST für “Lambda”-Test

```
1  import java.lang.Integer;
2
3  public class Lambda {
4
5    m () {
6     var lam1 = (x) -> {
7          return x;
8     };
9     return lam1;
10   }
11 }
```

Quelltext F.1: Quelldatei “Lambda.jav” für die Verwendung in Tests für den “Java TX”-Compiler

```
1  class Lambda {
2
3  Lambda(){
4      super(());
5    }
6    TPH KIZQ m(){
7      TPH KIZR lam1;
8      lam1 = (TPH KIZS x) -> {
9        return x;
10     };
11     return lam1;
```

```
12    }
13
14    Lambda(){
15      super(());
16    }
17
18  }
```

Quelltext F.2: Grafische Repräsentation des AST für das Programm in "Lambda.jav"

F.2 Quelltext für "PatternMatching"-Test

```
1  import java.lang.String;
2
3  record Point(int x, int y) {}
4  interface Shape {}
5  record ColoredPoint(Point pt, String color) {}
6  record Rectangle(ColoredPoint upperLeft, ColoredPoint lowerRight)
       ↪ implements Shape {}
7  sealed class Color permits Blue, Red {}
8  class Blue extends Color {}
9  class Red extends Color {}
10
11 class PatternMatching {
12 void printColorOfUpperLeftPoint(Shape shape)
13 {
14     switch (shape) {
15         case Rectangle(ColoredPoint(Point pt, String color),
               ↪ ColoredPoint lowerRight) -> System.out.println("x:␣" + pt
               ↪ .x() + "␣/␣color:␣" + color + "␣/␣lowerRight:␣" +
               ↪ lowerRight);
16         default -> System.out.println("not␣a␣rectangle");
17     };
18 }
```

```
19 }
```

Quelltext F.3: Quelldatei "PatternMatching.jav" für die Verwendung in Tests für den "Java TX"-Compiler

Literaturverzeichnis

[1] *Abstract Syntax Tree vs Parse Tree*. [Zugriffsdatum: 28.06.2023]. 2023. URL: https://www.geeksforgeeks.org/abstract-syntax-tree-vs-parse-tree/.

[2] Oracle and/or its affiliates. *Bounded Type Parameters (The Java Tutorials: Learning the Java Language, Generics (Updated))*. [Accessed 11-Jul-2023]. 2023. URL: https://docs.oracle.com/javase/tutorial/java/generics/bounded.html.

[3] Oracle Corporation and/or its affiliates. *JDK Releases*. URL: https://www.java.com/releases/ (besucht am 10. 03. 2023).

[4] Oracle Corporation and/or its affiliates. *Oracle Java SE Support Roadmap*. 2023. URL: https://www.oracle.com/java/technologies/java-se-support-roadmap.html (besucht am 13. 07. 2023).

[5] Alfred V Aho et al. *Compilers: principles, techniques and tools*. 2020. Kap. 1.

[6] Grzegorz Piwowarek et. al. *Assertions in JUnit 4 and JUnit 5*. 2023. URL: https://www.baeldung.com/junit-assertions (besucht am 17. 07. 2023).

[7] *ArrayList (Java Platform SE 8)*. URL: https://docs.oracle.com/javase/8/docs/api/java/util/ArrayList.html (besucht am 16. 03. 2023).

[8] Raghuram Bharathan. *Apache Maven Cookbook*. Packt Publishing Ltd, 2015.

[9] Gavin Bierman. *JEP 361: Switch Expressions*. Oracle Corporation and/or its affiliates. 2019. URL: https://openjdk.org/jeps/361 (besucht am 15. 07. 2023).

[10] Gavin Biermann. *JEP 406: Pattern Matching for switch (Preview)*. Oracle Corporation and/or its affiliates. 2018. URL: https://openjdk.org/jeps/406 (besucht am 16. 03. 2023).

[11] Gavin Biermann. *JEP 406: Pattern Matching for switch (Second Preview)*. Oracle Corporation and/or its affiliates. 2021. URL: https://openjdk.org/jeps/420 (besucht am 16. 03. 2023).

[12] Gavin Biermann. *JEP 406: Pattern Matching for switch (Third Preview)*. Oracle Corporation and/or its affiliates. 2022. URL: https://openjdk.org/jeps/427 (besucht am 16. 03. 2023).

[13] Gavin Biermann. *JEP 440: Record Patterns*. Oracle Corporation and/or its affiliates. 2023. URL: https://openjdk.org/jeps/441 (besucht am 15. 07. 2023).

[14] Gavin Biermann. *JEP 441: Pattern Matching for switch*. Oracle Corporation and/or its affiliates. 2023. URL: https://openjdk.org/jeps/441 (besucht am 16. 03. 2023).

[15] Gilad Bracha. „Generics in the Java Programming Language". In: (Jan. 2004).

[16] Gilad Bracha. *Generics in the Java programming language*. 2004.

[17] Bill Campbell, Swami Iyer und Bahar Akbal-Delibas. *Introduction to compiler construction in a Java world*. CRC Press, 2012. Kap. 1.

[18] Bill Campbell, Swami Iyer und Bahar Akbal-Delibas. *Introduction to compiler construction in a Java world*. CRC Press, 2012. Kap. Appendix D.

[19] Pierre Carbonnelle. *PYPL PopularitY of Programming Language*. 1. Apr. 2015. URL: https://pypl.github.io/PYPL.html (besucht am 03. 04. 2023).

[20] Michel Charpentier. *Functional and Concurrent Programming Core Concepts and Features.* Pearson Education (US), 2022. Kap. 15. ISBN: 978-0-13-746654-2. URL: https://login.ezproxy-dhma.redi-bw.de/login?url=https://search.ebscohost.com/login.aspx?direct=true&db=edsgvk&AN=edsgvk.1824484194&site=eds-live.

[21] Michel Charpentier. *Functional and Concurrent Programming Core Concepts and Features.* Pearson Education (US), 2022. Kap. 5. ISBN: 978-0-13-746654-2. URL: https://login.ezproxy-dhma.redi-bw.de/login?url=https://search.ebscohost.com/login.aspx?direct=true&db=edsgvk&AN=edsgvk.1824484194&site=eds-live.

[22] *Compilation and Execution of a Java Program*. GeeksforGeeks. Section: Java. 16. Apr. 2018. URL: https://www.geeksforgeeks.org/compilation-execution-java-program/ (besucht am 04. 06. 2023).

[23] Scala Docs contributors. *Tour of Scala - Type Inference*. URL: https://docs.scala-lang.org/tour/type-inference.html (besucht am 10. 03. 2023).

[24] Scala developers. *Pattern Matching*. 2023. URL: https://docs.scala-lang.org/tour/pattern-matching.html (besucht am 13. 07. 2023).

[25] *DIN 69901-2. Prozesse, Prozessmodell*. Deutsches Institut für Normung, 2010.

[26] Christa Dürscheid. *Syntax: Grundlagen und Theorien*. Bd. 3. Vandenhoeck & Ruprecht, 2007. Kap. 0.

[27] Friedrich Esser. *Front Matter*. München: Oldenbourg Wissenschaftsverlag, 2011. ISBN: 9783486710816. URL: https://doi.org/10.1524/9783486710816.fm.

[28] Free Software Foundation, Inc. *Host/Target specific installation notes for GCC - GNU Project*. URL: https://gcc.gnu.org/install/specific.html (besucht am 04. 06. 2023).

[29] Brian Goetz. *JEP 305: Pattern Matching for instanceof (Preview)*. Oracle Corporation and/or its affiliates. 2017. URL: https://openjdk.org/jeps/305 (besucht am 11. 03. 2023).

[30] Brian Goetz. *JEP 375: Pattern Matching for instanceof (Second Preview)*. Oracle Corporation and/or its affiliates. 2019. URL: https://openjdk.org/jeps/375 (besucht am 11. 03. 2023).

[31] Brian Goetz. *JEP 394: Pattern Matching for instanceof*. Oracle Corporation and/or its affiliates. 2020. URL: https://openjdk.org/jeps/394 (besucht am 11. 03. 2023).

[32] Brian Goetz. *JEP 395: Records*. Oracle Corporation and/or its affiliates. 2020. URL: https://openjdk.org/jeps/395 (besucht am 12. 07. 2023).

[33] Brian Goetz. *JEP 405: Record Patterns (Preview)*. Oracle Corporation and/or its affiliates. 2021. URL: https://openjdk.org/jeps/405 (besucht am 15. 03. 2023).

[34] Brian Goetz. *JEP 409: Sealed Classes*. Oracle Corporation and/or its affiliates. 2021. URL: https://openjdk.org/jeps/409 (besucht am 15. 03. 2023).

[35] Ralf Hartmut Güting und Stefan Dieker. *Datenstrukturen und Algorithmen*. Bd. 3. Springer.

[36] HaskellWiki. *Type inference — HaskellWiki*. 2007. URL: https://wiki.haskell.org/index.php?title=Type_inference&oldid=17047 (besucht am 10. 03. 2023).

[37] DHBW Stuttgart Campus Horb. *Forschungsschwerpunkt Typsysteme für objektorientierte Programmiersprachen*. URL: https://www.dhbw-stuttgart.de/horb/forschung-transfer/forschungsschwerpunkte/typsysteme-fuer-objektorientierte-programmiersprachen (besucht am 17. 02. 2023).

[38] Graham Hutton. *Programming in haskell*. Cambridge University Press, 2016.

[39] Michael Jäger. *Compilerbau - eine Einführung*. 2019. URL: https://homepages.thm.de/~hg52/lv/compiler/skripten/compilerskript/pdf/compilerskript.pdf.

[40] Donghoon Kim und Gangman Yi. „Measuring Syntactic Sugar Usage in Programming Languages: An Empirical Study of C# and Java Projects“. In: *Advances in Computer Science and its Applications*. Hrsg. von Hwa Young Jeong et al. Berlin, Heidelberg: Springer Berlin Heidelberg, 2014, S. 279–284. ISBN: 978-3-642-41674-3.

[41] Stephan Kleuker und Stephan Kleuker. *Qualitätssicherung durch Softwaretests: Vorgehensweisen und Werkzeuge zum Test von Java-Programmen*. Springer, 2013.

[42] Peter J. Landin. „The mechanical evaluation of expressions“. In: *The Computer Journal* (1964). DOI: doi:10.1093/comjnl/6.4.308.

[43] Tim Lindholm et al. „The Java® Virtual Machine Specification“. In: (2023). URL: https://docs.oracle.com/javase/specs/jvms/se20/jvms20.pdf (besucht am 04. 06. 2023).

[44] Miran Lipovača. *Syntax in Functions. Pattern Matching*. 2011. URL: http://learnyouahaskell.com/syntax-in-functions#pattern-matching (besucht am 16. 07. 2023).

[45] Martin Odersky, Lex Spoon und Bill Venners. *Scala*. 2011.

[46] Terence Parr. *The definitive ANTLR 4 reference / Terence Parr.* Pragmatic Programmers. The Pragmatic Programmers, 2014. URL: https://login.ezproxy-dhma.redi-bw.de/login?url=https://search.ebscohost.com/login.aspx?direct=true&db=cat08845a&AN=bkm.1680260898&lang=de&site=eds-live.

[47] B.C. Pierce. *Types and Programming Languages*. The MIT Press. MIT Press, 2002. ISBN: 9780262162098. URL: https://books.google.de/books?id=ti6zoAC9Ph8C.

[48] Martin Plümicke und Andreas Stadelmeier. „Introducing Scala-like Function Types into Java-TX“. In: *Proceedings of the 14th International Conference on Managed Languages and Runtimes*. ManLang 2017. Prague, Czech Republic: Association for Computing Machinery, 2017, S. 23–34. ISBN: 9781450353403. DOI: 10.1145/3132190.3132203. URL: https://doi.org/10.1145/3132190.3132203.

[49] Zink Plümicke. „Java-TX: The language“. In: DHBW Stuttgart, 2022. URL: https://www.dhbw-stuttgart.de/fileadmin/dateien/Forschung/Forschungsschwerpunkte_Technik/DHBW_Stuttgart_INSIGHTS_1_2022_Java-TX_The_language.pdf.

[50] Mark Reinhold. *JDK Enhancement-Proposal & Roadmap Process*. Oracle Corporation and/or its affiliates. 2018. URL: https://openjdk.org/jeps/1 (besucht am 13. 07. 2023).

[51] Mark Reinhold. *JEP Index*. Oracle Corporation and/or its affiliates. 2023. URL: https://openjdk.org/jeps/0 (besucht am 13. 07. 2023).

[52] Herbert Schildt. *Java : A Beginner's Guide, Ninth Edition, 9th Edition / Schildt, Herbert.* McGraw-Hill, 2022. ISBN: 9781260463569. URL: https://login.ezproxy-dhma.redi-bw.de/login?url=https://search.ebscohost.com/login.aspx?direct=true&db=cat08845a&AN=bkm.1785838474&lang=de&site=eds-live.

[53] Herbert Schildt. *Java : A Beginner's Guide, Ninth Edition, 9th Edition / Schildt, Herbert.* McGraw-Hill, 2022. Kap. 13. ISBN: 9781260463569. URL: https://login.ezproxy-dhma.redi-bw.de/login?url=https://search.ebscohost.com/login.aspx?direct=true&db=cat08845a&AN=bkm.1785838474&lang=de&site=eds-live.

[54] *The Java® Language Specification. Java SE 8 Edition*. 2015. URL: https://docs.oracle.com/javase/specs/jls/se8/jls8.pdf (besucht am 18. 03. 2023).

[55] Linda Torczon und Keith Cooper. *Engineering A Compiler*. 2nd. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2007. ISBN: 012088478X.

[56] *Type Inference*. URL: https://docs.oracle.com/javase/tutorial/java/generics/genTypeInference.html (besucht am 17. 03. 2023).

[57] Andrei Voronkov und Irina B. Virbitskaite, Hrsg. *Perspectives of System Informatics - 9th International Ershov Informatics Conference, PSI 2014, St. Petersburg, Russia, June 24-27, 2014. Revised Selected Papers*. Bd. 8974. Lecture Notes in Computer Science. Springer, 2015, S. 248–256. ISBN: 978-3-662-46822-7. DOI: 10.1007/978-3-662-46823-4. URL: https://doi.org/10.1007/978-3-662-46823-4.

[58] Gottfried Vossen und Kurt-Ulrich Witt. *Grundkurs Theoretische Informatik*. Springer, 2016.

[59] Wikipedia contributors. *Type inference — Wikipedia, The Free Encyclopedia*. 2023. URL: https://en.wikipedia.org/w/index.php?title=Type_inference&oldid=1131268898 (besucht am 13. 03. 2023).