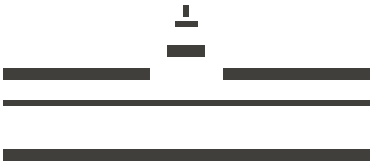

Westfälische
Wilhelms-Universität
Münster

Fachbereich für Mathematik und Informatik

Entwurf und Implementierung einer ADT-orientierten Programmiersprache

Autoren

Michael Burzan

Studiengang: M. Sc. Informatik

und

Michael Kußmann

Studiengang: Informatik (Diplom)

Gutachter: Prof. Dr. Achim Clausing, Prof. Dr. Markus Müller-Olm

Münster, Februar 2014

Inhaltsverzeichnis

Vorwort

Die vorliegende Arbeit wurde von uns – Michael Burzan und Michael Kußmann – als Gemeinschaftsprojekt konzipiert. Das heißt beide Autoren haben in gleichem Maße Anteil am Entwurf der Sprache und an ihrer Implementierung.

Aufgrund der gemeinsamen Arbeit ist eine klare Aufteilung der Arbeit nicht sinnvoll möglich. Deshalb wurde ein zusammenhängender Text verfasst, in dem jeder Autor jeweils einzelne Abschnitte verfasst und sich für diese verantwortlich zeigt. Dies geschieht durch Nennung der Initialen des jeweiligen Autors (MB bzw. MK) am Beginn des Texts. Dabei werden die Initialen nur dann angeführt, wenn die Autorenschaft wechselt. Zusätzlich sind die Seitenzahlen mit den jeweiligen Initialen markiert. Auf Seiten, auf denen die Autorenschaft wechselt, sind jedoch nur die Initialen ausschlaggebend. Für einige Abschnitte ist eine Aufteilung der Verantwortlichkeit nicht möglich bzw. sinnvoll. Diese werden durch Nennung der Initialen beider Autoren gekennzeichnet und haben keine Initialen an ihrer Seitenzahl.

Aufgrund der Anforderungen der Diplom- und Masterprüfungsordnung(en), die eine getrennte Abgabe der Abschlussarbeiten vorsehen, wird der gemeinsame Text für die Abgabe entsprechend der Autorenschaft aufgetrennt. Abschnitte für die sich beide Autoren verantwortlich zeigen, kommen, entsprechend markiert, in beiden Arbeiten vor. Die Urheberschaft des Programmcodes wurde, durch Nennung des jeweiligen Autors, im Quelltext kenntlich gemacht. Dabei wurden die einzelnen Klassen demjenigen Autor zugeordnet, der den größten Anteil an ihrer Implementierung hatte. In Einzelfällen werden beide Autoren genannt.

Als Vorbild für dieses Vorwort diente [SR03, S. 2f].

1. Einleitung

MB
MK

In dieser Arbeit wird die Entwicklung der am abstrakten Datentyp (ADT) orientierten Programmiersprache namens TeaJay dokumentiert und dem Leser werden grundlegende Ideen vermittelt, wie TeaJay in Lehre und Praxis eingesetzt werden kann. Dazu wurde im Zuge dieser Arbeit ein Prototyp-Compiler für TeaJay entwickelt.

1.1. Motivation und Zielsetzung

Programmiersprachen sind Sprachen und als solche Medien des Denkens [Cla11, S. 2]. Daher sollte der Entwurf einer Programmiersprache immer zum Ziel haben, den Programmierer beim Lösen von Problemen zu unterstützen, indem die Sprache ihm dabei hilft, das Problem zu strukturieren.[1] Dies gilt ganz besonders für Lehrsprachen, da Programmieranfänger in der Regel noch keine großen Erfahrungen darin haben, Probleme formal zu strukturieren. Daher sollte eine Lehrsprache mächtige Abstraktionsmechanismen besitzen, die sich gleichzeitig an der realen Welt orientieren. Ein Ansatz, diese beiden Anforderungen zu vereinen, ist die objektorientierte Programmierung, welche es dem Programmierer ermöglicht, Probleme durch Abbilden auf eine Klassenhierarchie zu strukturieren. Dabei definiert jede Klasse auch einen neuen Typ. Viele Programmiersprachen unterstützen die Erstellung von benutzerdefinierten Typen, jedoch erlauben sie meistens die Vermischung der Definition eines Typs mit seiner Implementierung oder erzwingen diese sogar. Spätestens beim Übersetzen werden Typ und Implementierung fest miteinander verbunden. Es gibt zwar in vielen Sprachen Bibliotheken, welche das dynamische Laden von Typen ermöglichen, jedoch sind diese häufig umständlich zu benutzen und können auch keinen Nutzen aus einer eventuell vorhandenen statischen Typprüfung ziehen. Zudem setzen viele Sprachen, wie z.B. Java oder C++, das Geheimnisprinzip nicht konsequent durch und erlauben z.B. den direkten Zugriff auf interne Zustände von Typen, was das Ersetzen einer Implementierung zusätzlich erschwert. TeaJays Typbegriff soll sich daher stark am Begriff des abstrakten Datentyps (ADT) orientieren und nur die Beschreibung einer Schnittstelle mit Sorten und Operationen darstellen. Die Kommunikation mit dem Typ soll ausschließlich über diese Schnittstelle geschehen und es soll möglich sein, die Implementierung des Typs auch nach der Kompilierzeit problemlos zu wechseln. Die Bindung zwischen Typ und Implementierung soll also so dynamisch wie möglich sein. Auch sollen in TeaJays Typsystem eingebaute und benutzerdefinierte Typen möglichst gleichberechtigt sein.

[1] Im Fall von Programmiersprachen bedeutet das, ein Modell zu entwickeln, welches sich auf den Rechner übersetzen lässt.

Wer eine objektorientierte Sprache beherrscht, weiß, dass zum Erzeugen einer Instanz einer Klasse (bzw. eines Typs) der konkrete Name der Klasse bekannt sein muss. TeaJay wird im Zuge der Trennung von Typ und Implementierung auch dies ändern: Um einen Typ erzeugen zu können, soll nur der Name des Typs bekannt sein müssen. Eine Implementierung muss erst dann vorliegen, wenn das Programm zur Ausführung kommt und kann dann dynamisch gesucht werden oder durch andere Mittel bekanntgegeben werden. Das soll dazu führen, dass es in TeaJay natürlich erscheint, modulare Programme zu schreiben.

Ziel dieser Arbeit ist es, eine objektorientierte Programmiersprache zu entwerfen und zu implementieren, die den Begriff des Typs (im Sinne eines ADTs) in den Vordergrund stellt und dadurch das Nachdenken über den Schnittstellenentwurf fördert. Dies soll einem Entwickler dabei helfen, Probleme zu strukturieren, indem die Sprache es natürlich erscheinen lässt, Programme in schwach gekoppelte Module zu unterteilen. Dabei muss ein Entwickler zwangsläufig über die Grenzen der einzelnen Module (Schnittstellen) nachdenken. TeaJay soll vorwiegend eine Lehrsprache sein, sich jedoch nicht so weit von der Pragmatik entfernen, dass sie nur zu einer Inselsprache der Lehre werden kann. Es soll auch möglich sein, verschiedene Programmierparadigmen in TeaJay zu lehren. TeaJay soll stark und statisch typisiert sein und das Typsystem soll so wenige Unregelmäßigkeiten wie möglich aufweisen. Dieser Aspekt soll vor allem der Lehre zugutekommen. Jedoch unterstützt ein mächtiges Typsystem auch erfahrene Entwickler, Fehler zu vermeiden. Des Weiteren soll TeaJay als festen Bestandteil der Sprache automatische Speicherbereinigung unterstützen.

1.2. Kapitelübersicht

Kapitel 1 enthält eine Einleitung und diese Kapitelübersicht.

Kapitel 2 diskutiert einige Grundlagen zu Programmiersprachen und abstrakten Datentypen. Zudem werden in diesem Kapitel einige Kriterien aufgestellt, welche, nach Meinung der Autoren, eine Lehrsprache erfüllen sollte.

Kapitel 3 beschreibt die Sprache TeaJay und diskutiert dabei, mit Fokus auf die Lehre, einige Entwurfsentscheidungen.

Kapitel 4 soll dem Leser einen tieferen Einblick in TeaJay erlauben und zeigt, vor allem anhand von Beispielen, auf, wie TeaJay genutzt werden kann. Der Lehraspekt von TeaJay wird in diesem Kapitel vertieft.

Kapitel 5 beschreibt die Implementierung eines Compilers und einer Laufzeitumgebung für TeaJay.

Kapitel 6 zeigt Möglichkeiten für eine Weiterentwicklung von TeaJay auf.

Kapitel 7 zieht ein Fazit.

2. Grundlagen

MB
MK

Heute existiert eine Vielzahl verschiedener Programmiersprachen. Bis auf wenige Ausnahmen fanden die Meisten keine weite Verbreitung. Jedoch wurden durch vielfältige Erschaffung von Programmiersprachen die Kenntnisse über allgemeine Konzepte des Entwurfs vergrößert. In diesem Kapitel werden die wichtigsten Konzepte und Eigenschaften, welche bekannte Programmiersprachen besitzen, kurz vorgestellt. Da Programmiersprachen gelernt werden müssen, bevor sie genutzt werden können, nennt dieses Kapitel einige für die Lehre geeignete Eigenschaften, die eine Lehrsprache haben sollte. Des Weiteren wird der Begriff abstrakter Datentyp erläutert und vier verschiedene Programmiersprachen werden zusammenfassend vorgestellt.

2.1. Programmierparadigmen

MB

Programmiersprachen können danach klassifiziert werden, welche Programmierparadigmen sie unterstützen. Dabei beschreibt ein Programmierparadigma, die Weise, wie ein Problem in einer Programmiersprache gelöst werden kann und gibt sie dadurch auch in Teilen vor. In der Literatur werden für gewöhnlich vier verschiedene Hauptparadigmen unterschieden: Diese sind das imperative, das objektorientierte, das funktionale und das logische Programmierparadigma. Es ist heutzutage durchaus üblich, dass eine Programmiersprache mehrere Paradigmen erfüllt, zum Beispiel existieren Sprachen, die objektorientiert und imperativ sind und zudem funktionale Elemente besitzen.

Das imperative Paradigma:
Programme werden als Folge von Anweisungen beschrieben, die sequenziell abgearbeitet werden. Die sequenzielle Ausführung von Anweisungen kann in imperativen Sprachen durch bedingte und unbedingte Sprünge verändert werden. Eng verbunden mit dem imperativen Paradigma sind Programmvariablen, die während der Ausführung eines Programms durch Anweisungen Zustandsänderungen durchleben können. Dies ist im Grunde der Kern des imperativen Paradigmas: Programme verändern Zustände. Moderne imperative Sprachen bieten neben Sprüngen als Kontrollstrukturen auch Schleifen und Prozeduren bzw. Funktionen an, um den Programmfluss zu steuern.

Das funktionale Paradigma:
Programme werden als Funktionen aufgefasst, die Eingaben in Ausgaben abbilden. Dabei erlaubt das funktionale Paradigma sehr kompakte Programme zu realisieren. Eine Funktion selbst wird dadurch realisiert, dass sie weitere Funktionen aufruft. Hierdurch hat das funktionale Paradigma den Begriff rekursives Programmieren geprägt. Weitere Funktionen können vom Nutzer

erzeugte Funktionen oder Operatoren sein, die auf die Eingabe der Funktion angewendet werden. Operatoren werden in den meisten funktionalen Programmiersprachen auch als Funktionen verstanden. Variablen kennt das funktionale Paradigma nicht, denn Zustandsänderungen werden nicht benötigt.

Das objektorientierte Paradigma:
Die meisten objektorientierten Programmiersprachen sind zustandsbasiert und somit verwandt mit [dem] imperativen [Paradigma] [BH98]. Während in imperativen Sprachen Variablen und Anweisungen, die Zustandsänderungen an der Variablen durchführen, als getrennt betrachtet werden, werden diese in objektorientierten Sprachen unter dem Begriff Objekt miteinander verschmolzen. Ein Objekt wird dabei als eine Struktur betrachtet, die Zustände verwaltet und Methoden besitzt, diese zu ändern. Die Manipulation eines Objekts wird als Nachrichtenaustausch zwischen Objekten verstanden. Eine Klasse wird als Abstraktion eines Objekts aufgefasst. Hierbei definiert die Klasse die notwendigen Attribute und Schnittstellen, die eine Instanz (Objekt) der Klasse besitzt.

In objektorientierten Sprachen ist es zudem möglich, Klassen in Hierarchien einzuordnen und so Verwandtschaftsbeziehungen zu modellieren. Dies ermöglicht es, einen hohen Grad an Abstraktion von Problemen durchzuführen.

Das logische Paradigma:
In Sprachen, die dem logischen Paradigma folgen, werden Programme geschrieben, die Fragen an den Computer darstellen (vgl. [Cla11]). Dieser gibt eine Antwort, die er mithilfe der Prädikatenlogik gewissermaßen selbständig findet [ebd.].

Weitere Paradigmen, die eine kurze Vorstellung verdienen, sind nachfolgend aufgelistet. Diese Paradigmen finden sich bis auf wenige Ausnahmen häufig als ergänzende Paradigmen in Programmiersprachen wieder.

Paradigma	Beschreibung
Parallele Programmierung	Die Sprache besitzt Eigenschaften, die es vereinfachen, Multithreading zu betreiben. Hierbei bietet die Sprache z.B. Monitore an, um die Synchronisation von Threads zu ermöglichen.
Constraint Programmierung	Die Sprache ermöglicht eine Erweiterung der logischen Programmierung um Ausdrücke wie mathematische Gleichungen
Datenflussprogrammierung	*In der Datenflussprogrammierung sind Programme als Netzwerk von Black-Box-Prozessen definiert, welche ihre Daten durch vordefinierte Kommunikation manipulieren und austauschen* [Mor].
Verteilte Programmierung	Die Sprache besitzt Eigenschaften, die es vereinfachen mit mehreren physisch getrennten Systemen zu kommunizieren und Daten mit ihnen auszutauschen.

Generische Programmierung	Entwickler haben die Möglichkeit, in der Sprache Algorithmen und Datenstrukturen unabhängig von konkreten Typen zu implementieren, so dass sie zum Zeitpunkt der Implementierung nicht bekannt sein müssen.
Metaprogrammierung	Die Sprache ermöglicht es *[Programmen, über sich selbst] Informationen zu sammeln und sich entsprechend neuer Anforderungen selbst zu modifizieren [MK06].*
Visuelle Programmierung	Diese Sprachen erlauben, Programme grafisch auszudrücken. Dies kann z.B. aussehen wie Programmablaufpläne, die aus der UML bekannt sind.

Tabelle 2.1.: weitere Programmierparadigmen

2.2. Typen und Typsysteme

Typen und Typsysteme bilden das Fundament einer jeden modernen Programmiersprache. Allerdings existieren auch Sprachen, die weder Typen noch ein Typsystem besitzen, dies sind z.B. Assemblersprachen. Dabei spezifizieren Typen, welche Operationen angewendet werden dürfen und welche nicht. Der Beispielcode in Abbildung 2.1 zeigt eine falsche Verwendung von Operationen in einer getypten und einer ungetypten Sprache. Während in der getypten Sprache der letzte Ausdruck als sinnlos

```
Integer j = 0
String c = "HALLO"
j = j - 1
j = c - j
```

```
j = 0
c = "HALLO"
j = j - 1
j = c - j
```

Abbildung 2.1.: links: getypte Version; rechts: ungetypte Version

bzw. als falsch erkannt wird, wird er in der ungetypen Sprache ausgewertet. Dabei stellt sich die Frage, welchen Wert j hat, nachdem der Ausdruck ausgewertet wurde. Die wahrscheinlichste Antwort ist, dass das Bitmuster von c minus j gerechnet wird. Es ist aber mit Sicherheit nicht das, was der Entwickler beabsichtigt hat. Getypte Sprachen werden kritisiert, weil sie vermeintlich die Entwickler bevormunden und zu ineffizienten Programmen führen. Strom entkräftet diese Argumente und weist darauf hin, dass moderne Compiler besseren und effektiveren Code erzeugen können als die meisten Entwickler und dass die Bevormundung ein kleiner Preis für die Zeitersparnis ist, die sich durch getypte Sprachen erzielen lässt (vgl. [Str96]).

Hochsprachen erlauben im Allgemeinen, komplexe aus primitiven Typen zu bauen. Primitive Typen sind solche, die in der Sprache eingebaut sind und Grundoperationen erlauben. Die häufigsten primitiven Typen sind **Integer** für ganzzahlige Ausdrücke, **Double** für Fließkommaoperationen, **Character** für ASCII-Zeichenmanipulationen, **Boolean** für Wahrheitswerte und **Void** für den leeren Aus-

druck. Die primitiven Typen sind die Atome ihrer Sprache und können nicht weiter zerlegt werden. Damit komplexe Typen gebaut werden können, muss die Programmiersprache Behälter zur Aggregation besitzen. Die Einfachsten sind:

- Strukturen bzw. Verbunde (Records): Sie fassen eine konkrete Anzahl von Typen zusammen und geben diesen einen neuen Typnamen.
- Felder: Sie definieren eine feste Indexstruktur über einen einzigen Typ und erlauben den Zugriff über einen Index auf die Elemente des Feldes.
- Listen: Sie verwalten beliebig viele Typen, erlauben den Zugriff aber nicht über einen Index, sondern durch Iteration über den Beginn der Liste.

Eine weitere Art von primitiven Typen sind **Zeiger** bzw. **Referenzen**. Sie dienen als Verweise auf Adressen bzw. auf den Inhalt einer Adresse im Speicher. Diese Art von Typ lässt sich auch dafür nutzen, komplexe Typen zu konstruieren. Es ist z.B. möglich, durch Verkettung von Zeigern oder Referenzen eine Liste zu implementieren.

Damit in Programmiersprachen Typen eingesetzt und falsche Operationen entdeckt werden können, muss ein Typsystem entworfen und entwickelt werden. Ein Typsystem ist das Regelwerk, nach dem entschieden wird, ob Operationen auf einen bestimmten Typ erlaubt sind oder nicht. Dieses Regelwerk wird zur Laufzeit oder Kompilierzeit eines Programms angewendet. Findet die Anwendung des Regelwerks zur Kompilierzeit statt, wird von einer statischen Typisierung gesprochen und im Fall der Laufzeit von dynamischer Typisierung. Der Unterschied zwischen den beiden Typisierungen liegt im Zeitpunkt des Fehlerauftritts. Bei statisch typisierten Sprachen findet dies während der Kompilierzeit und in dynamisch typisierten Sprachen während der Laufzeit statt.

Typisierte Sprachen werden zudem in stark oder schwach typisiert unterteilt. Dabei sind schwach typisierte Sprachen dazu in der Lage nicht-konforme Operationen auf Typen anzuwenden, indem implizit oder explizit ein anderer Typ für den Ausdruck angenommen wird. Bei stark typisierten Sprachen existieren solche Möglichkeiten nur eingeschränkt. Diese Definition soll nicht als endgültig betrachtet werden. Da die meisten Programmiersprachen auch pragmatischen Entwurfsentscheidungen unterliegen, erlauben Sprachen, die als stark typisiert gelten, Ausnahmen unter bestimmten Bedingungen. Im Fall von objektorientierten Programmiersprachen, in denen Verwandtschaftsbeziehungen beachtet werden, muss ein starkes Typsystem dies berücksichtigen, da es ansonsten nicht möglich ist, allgemeinere Objekte durch spezielle auszutauschen. Solch ein Typsystem wird polymorph genannt. Ist ein Typsystem polymorph, so hat eine Variable meist drei verschiedene Typen, dies sind (vgl. [Pun07, S.18]):

Deklarierter Typ:
Insofern in einer Sprache Variablen deklariert werden, ist er der Typ der Variablen, der während der Deklaration angegeben wurde.

Statischer Typ:
Der Typ wird statisch vom Compiler ermittelt und kann spezifischer sein als der deklarierte Typ.

Dynamischer Typ:
Dies ist der Laufzeittyp einer Variablen und kann sich bei jeder Zuweisung ändern.

Einige Programmiersprachen gestatten generische Typen zu entwerfen und zu nutzen. Dabei führt die Generizität in ein polymorphes Typsystem die Möglichkeit ein, Typen und damit ihre Schnittstelle unabhängig von konkreten Typen zu gestalten. Hierfür werden die generischen Typparameter genutzt. Diese sind Stellvertreter für konkrete Typen. Dabei beschreiben die generischen Typparameter die geforderte Schnittstelle, die ein konkreter Typ besitzen muss, damit die Schnittstelle des generischen Typs mit ihr umgehen kann. In der Implementierung einer Schnittstelle wird zur Kompilierzeit nur mit dem Stellvertreter gearbeitet. Hinzu kommt, dass generische Typen einer besonderen Behandlung im Typsystem unterzogen werden, denn wird ein generischer Typ mit einem konkreten Typen erzeugt, so ist die Schnittstelle des generischen Typs auf den angegebenen konkreten Typ umgestellt. Das bedeutet, es müssen keine Typumwandlungen seitens des Entwicklers durchgeführt werden. Hierdurch wird eine höhere Flexibilität erreicht als das polymorphe Typsystem erlaubt.

Die obige Beschreibung erläutert dabei die homogene Übersetzung von Generizität, wie sie in Java existiert (vgl. [Pun07, S.111f.]). Eine andere Form ist die heterogene Übersetzung, diese ähnelt dem Konzept von Copy und Paste. Da jede Erzeugung einer Instanz eines generischen Typs dazu führt, dass der Compiler einen neuen Typ erzeugt, in dem jedes Vorkommen des generischen Parameters durch den bei der Instanzerzeugung angegebenen Typ ersetzt wird (vgl. [ebd. S.112]).

2.3. Abstrakte Datentypen

MB
MK

Programmiersprachen, die Verbunde und Prozeduren bzw. Funktionen besitzen, erlauben meist abstrakte Datentypen zu definieren. ADTs werden auch benutzerdefinierte Datentypen genannt, da sie es ermöglichen, Typen zu definieren, die den eingebauten Typen der Programmiersprache ähneln (vgl. [Coo]). Das bedeutet, ADTs sind eine Erweiterung des Typsystems, so dass für sie dasselbe Regelsystem Anwendung findet. Damit dies geleistet werden kann, wird ein ADT nur über seine erlaubten Operationen spezifiziert. So sind alle Implementierungsdetails dem Klienten verborgen. Dieser Sachverhalt wird als **Information Hiding** oder Geheimnisprinzip bezeichnet. Ein Vorteil bei konsequenter Trennung von Implementierung und Schnittstelle ist der erleichterte Austausch von Implementierungen ohne weitere Änderungen des Klienten. Ein Beispiel für einen ADT ist in Abbildung 2.2 gezeigt.

Wenn Klienten die Schnittstelle des ADTs **IntStack** verwenden wollen, muss den Entwicklern mitgeteilt werden, wie sie korrekt genutzt wird. Des Weiteren ist es nicht immer gegeben, dass der Entwerfer der Schnittstelle auch ihr Implementierer ist. Das heißt, es muss kenntlich gemacht werden, was unter einem korrekten Verhalten des ADTs verstanden wird. Hierfür gibt es zwei unterschiedliche Konzepte. Das eine Konzept ist die Kommentierung der Schnittstelle und das andere die formale Spezifikation, wie sie von Guttag (vgl. [Gut77]) gezeigt wird. Bei der Kommentierung ist festzuhalten, dass sie das Verhalten der Schnittstelle beschreibt, aber Missverständ-

```
Struct IntStack {
List_of_Integer stack;
}
IntStack create() {...}
Void push(IntStack s){...}
Integer top(IntStack s) {...}
Void pop(IntStack s){...}
Boolean isEmpty(IntStack s){...}
```

```
Struct IntStack;
IntStack create();
Void push(IntStack s);
Integer top(IntStack s);
Void pop(IntStack s);
Boolean isEmpty(IntStack s);
```

Abbildung 2.2.: links: Implementierung; rechts: Schnittstelle

nisse oder das gänzliche Ignorieren des Implementierers nicht ausschließen kann. Jedoch ist es denkbar, dass der Entwerfer des ADTs einen Testfall mitliefert. Dieser kann testen, ob die Implementierung die geforderte Spezifikation erfüllt. Durch die formale Spezifikation können moderne Compiler unter bestimmten Bedingungen dem Implementierer darüber Rückmeldungen geben, ob er die Spezifikation erfüllt.

ADTs lassen sich durch Klassen definieren, denn eine Klasse kann die Implementierungsdetails verstecken und eine öffentliche Schnittstelle definieren. Diese Schnittstelle definiert dann einen ADT. Allerdings erzwingen die meisten objektorientierten Sprachen das Geheimnisprinzip nicht. So muss der Entwickler dies durch diszipliniertes Verhalten leisten. Klassen sind zudem in den meisten Fällen Mitglieder einer Vererbungshierarchie und können somit auf interne Informationen der Klasse, von der sie erben, zugreifen und diese nach Belieben ändern. Hierdurch weichen die meisten objektorientierten Sprachen das Geheimnisprinzip auf.

2.4. Programmiersprachen in der Lehre

Programmiersprachen sind Werkzeuge, mit denen Probleme gelöst werden sollen. Doch sie sind auch Mittel der Kommunikation der Gedanken von Entwicklern. Das Lehren von Programmiersprachen in Schule und Universität unterscheidet sich grundlegend vom Lehren einer natürlichen Sprache. Das Ziel ist hierbei nicht die Vermittlung einer oder mehrerer Programmiersprachen an sich, sondern die Vermittlung von Fähigkeiten und Kenntnissen, die dazu geeignet sind, ein Problem in einer beliebigen Programmiersprache zu lösen (vgl. [Goe97]). Hierbei liegt für den Lehrenden die Herausforderung darin, zu entscheiden, welche Kompetenzen und Kenntnisse wichtig sind und welche nicht.

Eine einzelne Programmiersprache kann wahrscheinlich nicht als Lehrsprache für alle Kompetenzen, die der Lehrende als wichtig ansieht, dienen. Hieraus ergibt sich, dass es *die* Lehrsprache wohl nicht geben kann. Wenn die Lernenden beispielsweise rekursives Programmieren erlernen sollen, ist eine funktionale Sprache besser dazu geeignet als eine Multiparadigmensprache. Will der Lehrende hingegen das Erkennen und Anwenden für den sinnvollen Einsatz von Abstraktion fördern, sind objektorientierte Sprachen vermutlich geeigneter als funktionale.

Aus den Zielvorgaben des Landes Nordrhein-Westfalen für den Informatikunterricht der gymnasialen Oberstufe ergibt sich, dass die Schüler imperative, funktionale, logische und objektorientierte Programmierung erlernen sollen (vgl. [Cur99]). Dies

zu leisten ist ein ambitioniertes Ziel und deshalb sollte die Auswahl von Programmiersprachen sorgfältig vom Lehrenden durchdacht werden.

Für eine Programmiersprache, die in der Lehre eingesetzt wird, ist festzuhalten, dass sie die Modellierung von Phänomenen der realen Welt erleichtern und dieses nicht durch eine komplexe Notation behindert sollte (vgl. [Goe97]). Eine Forderung an eine Lehrsprache kann sein, dass sie eine weite Verbreitung in der Wirtschaft hat und viele Werkzeuge besitzt, die das Entwickeln unterstützen. Ist aber eine Sprache weit verbreitet, so sind viele Standardprobleme, die als Lerngegenstände dienen könnten, bereits gelöst. Somit entsteht für die Lehrenden und Lernenden ein Motivationsproblem, dem nur schwer etwas entgegengesetzt werden kann. Hilfsmittel bei der Entwicklung sind im Allgemeinen gut, aber es ist darauf zu achten, dass sie nicht zur Richtschnur der Lehre werden (vgl. [Cur99]). Weiterhin ist eine Programmiersprache als Lehrsprache, nach Goedicke und Strom, um so besser geeignet, je früher Fehler gefunden werden (vgl. [Goe97] [Str96]). Somit ist eine wichtige Forderung an eine Lehrsprache, dass sie statisch und stark typisiert ist. Ketz und Hug fassen zusammen, dass eine Lehrsprache folgende Eigenschaften besitzen soll: *schlanke, saubere, sichere, statisch typisierte, höhere Programmiersprache, die sich dazu eignet, modulare und objektorientierte Programmierung zu lehren und zu lernen* [H.K98]. Allerdings wird keine Angabe darüber gemacht, was die einzelnen Punkte genau bedeuten. In dieser Arbeit werden folgende Eigenschaften angenommen, die eine Programmiersprache als Lehrsprache auszeichnet:

- **statische Typisierung:**
  Dies dient dazu, die Fokussierung auf das Problem zu verstärken und das Ausschließen potenzieller Probleme zu ermöglichen.

- **Programme werden objektorientiert und modular implementiert:**
  Programmieranfänger sollen objektorientierte Konzepte verstehen und anwenden lernen, weil diese sich durchgesetzt haben. Die Möglichkeit, Programme modular aufzubauen dient, in einer Lehrsprache vor allem dem Zweck, zu verstehen wie Quellcode organisiert wird, um z.B. mit einer Lerngruppe ein Softwareprojekt durchzuführen.

- **imperative Programmierung mit funktionalen Elementen:**
  Hierdurch wird es ermöglicht, dass die Programmieranfänger die beiden verbreitetsten Arten, Programme zu formulieren, kennenlernen können.

- **vollständiges Bewahren des Geheimnisprinzips:**
  Durch die Bewahrung des Geheimnisprinzips soll gewährleistet werden, dass Anfänger über die Schnittstellen nachdenken um damit die Fokussierung auf den Entwurf von Modulen und Typen zu verstärken.

- **Prinzip der geringsten Überraschung:**
  Dieses Prinzip soll Anfängern bei der Fehlervermeidung helfen und sie dabei unterstützen, die richtigen Annahmen über die Semantik der Sprache zu entwickeln.

- **schlanker Satz an existierenden Hilfsmitteln:**
  Hierdurch wird es ermöglicht, dass Aufgaben, die gestellt werden, für den Lernenden sinnvoll erscheinen.
- **syntaktische Einheitlichkeit vor Mehrdeutigkeit:**
  Anfänger sollen ihre Lösungen auf einfache und eindeutige Weise erzeugen können. So werden sie durch verschiedene syntaktische aber gleichwertige semantische Elemente in der Sprache nicht abgelenkt.
- **automatisches Speichermanagement:**
  Wie bei der statischen Typsicherheit soll das Speichermanagement dem Nutzer abgenommen werden, damit dieser sich auf die Lösung der Problemstellung konzentrieren kann.

2.5. Vorstellung verschiedener Programmiersprachen

Die Entwicklung einer Programmiersprache kann nicht für sich isoliert betrachtet werden, da frühere Programmiersprachen häufig die neu Entwickelten beeinflusst haben. Die Entwickler betrachten dazu die Syntax von Programmiersprachen und nutzen diese für den Entwurf ihrer eigenen. Weiterhin gibt es universelle Konzepte, die sich durchgesetzt haben und um eigene neue erweitert werden.

Die meisten Programmiersprachen wurden nicht entwickelt, um speziell in der Lehre eingesetzt zu werden, sondern um Problemstellungen zu lösen. Deshalb kann nicht erwartet werden, dass eine der hier vorgestellten Programmiersprachen die oben genannten Kriterien erfüllt. Allerdings kann eine Sprache, die möglichst viele Eigenschaften erfüllt ein guter Ausgangspunkt für die Entwicklung sein.

2.5.1. Die Programmiersprache C

MB

Die Programmiersprache, die sehr stark den Entwurf vieler Programmiersprachen beeinflusste, ist C. Sie wurde in erster Linie dazu entworfen, um systemnah zu entwickeln. Weiterhin überlässt sie dem Entwickler alle Freiheiten, die mit ihrer Syntax möglich sind. Wenn z.B. andere Programmiersprachen die Indexierung durch eine negative Konstante als Fehler melden, ist es der Programmiersprache C im besten Fall, je nach Compilerhersteller, eine Warnung wert. Eine weitere Eigenschaft, die Anfängern Probleme bereitet, ist das Speichermanagement. Es muss komplett vom Entwickler übernommen werden. Anfänger verbringen viel Zeit damit, ihren Quellcode nach Speicherproblemen zu untersuchen. Die Programmiersprache C bietet Makros an die auf reiner Textersetzung, ohne Logik dahinter, basieren. Dies führt zu Effekten und Fehlern, die der Entwickler nicht beabsichtigt und die schwer lokalisierbar sind. Dies geschieht z.B. bei der Ermittlung des Minimums zweier Zahlen durch ein Makro:

```
1 #define min(a,b) a < b ? a : b
```

Dieses Makro ermittelt das Minimum aber bei einem Aufruf in dieser Form:

```
1 c = min(a++,b++);
```

So tritt durch die reine Textersetzung der Effekt auf, dass a oder b um zwei erhöht wurde, nicht um eins.

Die Sprache bietet die Möglichkeit, Datentypen in Records bzw. Strukturen zusammenzufassen. Modulares Programmieren wird durch das Makrokonzept in C ermöglicht. Dadurch lassen sich Daten zunächst vor dem User verstecken. Das generische Programmieren wird in C durch den speziellen Datentyp **void*** erreicht.

Durch das Nutzen von Strukturen und der modularen Programmierung ist es möglich, in C ADTs zu entwerfen und zu entwickeln. Das Beispiel in Abbildung 2.3 zeigt eine Möglichkeit. Wie dort zu sehen ist, lassen sich ADTs in C durch Kom-

```
#ifndef STACK_H
#define STACK_H

struct Stack;

// Erzeugen eines leeren Stacks
struct Stack * create();
// Element in den Stack hinzfügen
void push(struct Stack * s,void * element, size_t size);
// Oberstes Element vom Stack entfernen
// Falls Stack leer ist, wird NULL zurück gegeben
void * pop(struct Stack * s);
//Prüfen, ob der Stack leer ist
int isEmpty(struct Stack * s);

#endif
```

Quellcode 2.1: stack.h

```
#include "stack.h"

struct Stack{
  int elemCnt;
  int size;
  void ** array;
};

struct Stack * create(){}
void push(struct Stack * s,void * element,size_t size){}
void * pop(struct Stack * s){ }
int isEmpty(struct Stack * s){ }

void resize(struct Stack *s){ }
```

Quellcode 2.2: stack.c

Abbildung 2.3.: ADT Stack in C: oben .h Datei, unten .c Datei

mentare spezifizieren. Das Geheimnisprinzip wird durch die Aufteilung in Header- und C-Datei geleistet. Ein Nutzer kann die Headerdatei nutzen, um einen Stack zu erzeugen und mit diesem zu kommunizieren. Die interne Struktur des Stacks ist vor dem Nutzer versteckt.

Die Programmiersprache C ist für Anfänger zu komplex, denn ihre Semantik lässt zu viele Ausnahmen zu. Es gibt kein automatisches Speichermanagement und die Syntax von C ist mehrdeutig. Zudem ist C schwach typisiert, hierdurch sind Abfragen nötig, um die Typsicherheit zu garantieren. In Abbildung 2.3 erwartet die Push-Funktion ein void* und die Größe des Elements, welches dem Stack hinzugefügt werden soll. Diese Art der generischen Programmierung ist umständlich und fehleranfällig, da in dieser Methode der Speicher in der richtigen Größe alloziert und in korrekter Weise kopiert werden muss. Des Weiteren werden objektorientierte Konzepte in der Sprache nicht unterstützt. Diese Eigenschaften erschweren es C zu lehren und lernen.

2.5.2. Die Programmiersprache Java

Java fand in den letzten 15 Jahren weite Verbreitung in Lehre und Wirtschaft. Dies ist den Tatsachen geschuldet, dass diese Sprache als mächtig und leicht erlernbar gilt. Sie unterstützt eine Vielzahl von Paradigmen, darunter das imperative und objektorientierte. Sie besitzt zusätzlich Elemente des funktionalen Paradigmas. Des Weiteren unterstützt sie generisches, paralleles und verteiltes Programmieren. Die Programmiersprache Java besitzt ein starkes Typsystem und der syntaktische Sprachumfang ist schlank gehalten. Als Ziel setzt sich diese Sprache soweit wie möglich plattformunabhängig zu sein. Das heißt, Quellcode wird kompiliert und läuft dann auf allen Plattformen. Hierbei erzeugt der Java-Compiler in der Regel keinen ausführbaren Maschinencode sondern Bytecode. Der erzeugte Bytecode läuft in einer virtuellen Maschine, die mit Java mitgeliefert wird. Die virtuelle Maschine lässt sich für den Bytecode als Abstraktion der Hardware auffassen. Aus dieser Eigenschaft folgt, dass Java nur auf Systemen läuft, für die die Java Virtual Machine implementiert wurde.

In Java können ADTs mithilfe des Interface- und des objektorientierten Konzepts erstellt werden. Das Beispiel in Abbildung 2.4 zeigt diese Möglichkeit.

```
1  interface IStack<T>{
2    //Falls der Stack leer ist, gib true ↩
       ↪zurueck
3    //Ansonsten false
4    boolean isEmpty();
5    //Fuegt ein element in den Stack ein
6    void push(T element);
7    //Entfernt das oberste Element vom ↩
       ↪Stack
8    //Falls Stack leer wirft er eine ↩
       ↪Exception
9    T pop() throws Exception;
10 }
```

Quellcode 2.3: istack.java

```
1  class Stack<T> implements ↩
     ↪IStack<T>{
2
3    public Stack(){ }
4    public void push(T element) {↩
       ↪ }
5    public T pop()throws ↩
       ↪Exception{ }
6    public boolean isEmpty() { }
7
8  }
```

Quellcode 2.4: stack.java

Abbildung 2.4.: ADT Stack in Java: links Interface, rechts Klasse

Wie in Abbildung 2.4 zu sehen ist, lassen sich ADTs wie in C durch Kommentare spezifizieren. Durch das Aufteilen in Interface und Class ist die interne Struktur vor dem Nutzer versteckt. Allerdings ist es nicht möglich, in einem Interface anzugeben, wie ein Typ erzeugt wird [Jia].

Java ermöglicht es den Lernenden, sich stark auf die Lösung eines Problems zu konzentrieren. Dies wird durch das starke Typsystem, das automatische Speichermanagement und durch gute Fehlermeldungen des Compilers ermöglicht. Eine weitere Stärke für die Lehre ist, dass Java eine aktive Community hat, die viele Werkzeuge entwickelt, welche das Entwickeln erleichtern. Durch Javas weite Verbreitung findet ein Lernender auch schnell Hilfe.

Trotz der genannten Vorteile hat Java Probleme im Typsystem. So existiert unter anderem ein Fehler bei der statischen Typisierung, der im nachfolgenden Beispiel gezeigt wird:

```
1 Object[] array = new String[2]; //Diese Zuweisung dürfte nicht ↩
    ↪erlaubt werden
2 array[0] = new Integer(1);
3 array[1] = new Double(1.5);
```

Dieser Quellcode dürfte vom Java Compiler nicht akzeptiert werden, dennoch tut er es. Hierdurch tritt das Problem erst zur Laufzeit auf. Durch konsequentes Nutzen des Generikkonzepts und der mitgelieferten Bibliotheken ist es möglich, dieses Problem zu umgehen. Die benutzerdefinierten Typen wirken allerdings den Java Typen etwas fremd, da es es dem Typ String z.B. erlaubt ist, mit dem +-Operator die Konkatenation durchzuführen. Mit den Typen Integer, Float, Long und Double können alle Arithmetischen Operatoren ausgeführt werden. Für benutzerdefinierte Typen existieren solche Fähigkeiten nicht.

2.5.3. Die Programmiersprache Pascal

MK

Die Programmiersprache Pascal wurde 1970 von Niklaus Wirth als minimalistische imperative Lehrsprache entworfen (vgl. [Cla11, S. 129]). Pascal ist stark und statisch typisiert und das Typsystem von Pascal bietet einige Besonderheiten (vgl. [Cat13]):

- Pascal ermöglicht es, eigene Datentypen durch Einschränkung des Wertebereichs eingebauter Typen (vor allem ganzer Zahlen) zu definieren: Zum Beispiel wird mit **type byte** = 0..255; der Typ byte als vorzeichenlose 8-Bit Ganzzahl definiert. Man kann für Variablen auch direkt den Wertebereich angeben: **var** i : 0..42;. Hier hat i dann einen anonymen Typ, dessen Wertebereich $[0, 42] \subset \mathbb{Z}$ entspricht. Leider setzt der weit verbreitete Free Pascal Compiler [www13] standardmäßig viele Wertbereichseinschränkungen nicht hart durch:

```
1  program fpc;
2
3  var
4    small : 0..100;
5    big : 1024..2048;
6  begin
7    small := 111; {nur Warnung}
8    writeln(small); {Ausgabe: 111}
9    big := 1024;
10   small := big; {kein Kompilerfehler, keine Warnung}
11   writeln(small); {Ausgabe: 0}
12 end.
```

Quellcode 2.5: fpc.pas

- Pascal besitzt die Möglichkeit eigene Mengen ganzer Zahlen als Typ zu definieren: **type** digits = **set of** 0..9; **var** MyDigits : digit ; Mit diesen Definitionen ist nun folgender Quellcode gültig:

```
1     MyDigits := [2,3,8];
2     if 4 in MyDigits then
3       writeln('kann nicht passieren');
4     if 6 in [0..4, 6..8] then
5       writeln('klappt');
```

- Die Größe eines Arrays wird von Pascals Typsystem erfasst:

```
1     var arr : array[0..1] of integer;
2     begin
3       arr[0] := 1; arr[1] := 2;
4       writeln(arr[4]); {erzeugt eine Warnung}
5     end.
```

- Es existiert ein eingebauter Typ file, welcher den typsicheren Umgang mit Dateien erlaubt. Zum Beispiel kann man eine Datei so deklarieren, dass ihr Inhalt als eine Liste von ganzen Zahlen interpretiert wird:

```
1     type
2       intlist : array[0..10] of integer;
3     var
4       intfile : file of intlist;
```

- Zeiger werden von Pascal unterstützt, jedoch gibt es keine Zeigerarithmetik wie z.B. in C.

Pascal bietet zudem Aufzählungs- und Arraytypen, sowie zusammengesetzte Typen (record) als Mittel zur Aggregation. Ein Beispiel für zusammengesetzte Typen in Pascal sieht wie folgt aus:

```
1  program _record;
2
3  type person = record
4      name : string;
5      age : 0..130
6    end;
7  var Andy : person;
8  begin
9    with(Andy) do
10   begin
11     name := 'Andy';
12     age := 23;
13   end;
14   writeln(Andy.name); {Ausgabe: Andy}
15 end.
```

Quellcode 2.6: record.pas

Das ursprüngliche Pascal hat den Entwickler kaum bei der Definition eigener ADTs unterstützt. Es ist zwar in großem Umfang möglich neue Strukturen zu definieren, jedoch fehlt ein eigener Namensraum für dessen Operationen. Dieser Mangel wurde jedoch später durch Units (siehe [Ola13]) behoben. Units bilden einen eigenen Namensraum und erlauben es dem Entwickler eine öffentliche Schnittstelle zu definieren. Quellcode 2.7 zeigt dies am Beispiel eines Stacks.

```
1  unit Stapel;
2  { ein Stapel zur Speicherung von Zeichen }
3  interface {oeffentliche Schnittstelle}
4    type
5    tNatZahlNull = 0..maxint;
6    function isEmpty : boolean;
7    { liefert true, wenn der Stapel leer ist,
8      sonst false }
9    procedure top (var outElem : char);
10   { liefert das oberste Stapelelement }
11   procedure push (inElem : char);
12   { stapelt ein neues Element }
13   procedure pop;
14   { entfernt das oberste Stapelelement }
15
16 implementation
17   type
18   tRefStapelelem = ^tStapelelem;
19   tStapelelem = record
20                   info : char;
21                   next : tRefStapelelem
22                 end;
23   var
24   Anfang : tRefStapelelem;
25   { Anfangszeiger des Stapels }
26   procedure create;
27   { erzeugt den leeren Stapel }
28   begin
29     Anfang := nil
30   end; { create }
31   function isEmpty : boolean;
32   { liefert true, wenn der Stapel leer ist,
33     sonst false }
34   begin
35     isEmpty := (Anfang = nil)
36   end; { isEmpty }
37   procedure top (var outElem : char);
38   { liefert das oberste Stapelelement }
39   begin
40     if isEmpty then
41       writeln ('Stapel ist leer.')
42     else
43       outElem := Anfang^.info
44   end; { top }
45   procedure push (inElem : char);
46   { stapelt ein neues Element }
47     var
48     neu : tRefStapelelem;
49   begin
50     new (neu);
51     neu^.next := Anfang;
52     neu^.info := inElem;
53     Anfang := neu
54   end; { push }
55   procedure pop;
56   { entfernt das oberste Stapelelement }
57     var
58     hilf : tRefStapelelem;
59   begin
60     if isEmpty then
61       writeln ('Stapel ist leer.')
62     else
63     begin
64       hilf := Anfang;
65       Anfang := Anfang^.next;
66       dispose (hilf) { Keine automatische Speicherverwaltung fuer Referenzen }
67     end
68   end; { pop }
69 begin { Initialisierung }
70   create
71 end. { Stapel }
```

Quellcode 2.7: Implementierung eines Stacks in Pascal. Das Template für den Stack stammt aus einer Übungsaufgabe der Fernuniversität Hagen.

Das mächtige Typsystem von Pascal hat seine Vorzüge, wenn es darum geht sicheren (und verifizierbaren) Code zu schreiben. Es ist jedoch auch sehr umfangreich und anfangs schwer zu erlernen. Zudem könnte die nach heutigen Maßstäben ungewöhnliche Syntax von Pascal Anfänger zuerst abschrecken.

2.5.4. Die Programmiersprache Lisp

Lisp ist eine listenbasierte und funktionale Sprache. Der Name Lisp (**Lis**t **p**rocessing) beruht auf der Eigenschaft, dass die Sprache zur Aggregation eine unveränderliche[1] Liste bietet. Die meisten der oben genannten Sprachen bieten dagegen eine eher schwache Abstraktion des von-Neumann-Speichermodells an (Array). Eine Besonderheit von [t.]Lisp (vgl. [Cla11, S. 45]) ist, dass Funktionen und sogar eingebaute Operatoren auf natürliche Weise Typen der Sprache sind (*first-class citizen*). Wie viele interpretierte Sprachen besitzt auch t.Lisp eine rein dynamische Typisierung[2]. Dies hat zur Folge, dass Typfehler erst zur Laufzeit sichtbar werden und dann möglicherweise schwieriger zu identifizieren sind. Es vereinfacht aber auch das Entwickeln kleiner Programme.

Ein Vorteil funktionaler Sprachen, wie Lisp, ist es, dass eine ADT-Spezifikation zumeist auch gleich seine Implementierung ist. Andererseits bietet Lisp keine Namensräume und der eigentlich interne Zustand eines ADTs muss vom Klienten selber verwaltet werden. Dies wird in Quellcode 2.8 deutlich: Jedes push erzeugt eine neue Liste (welche den Stack repräsentiert) und muss vom Klienten beim nächsten push übergeben werden. Quellcode 2.8 wird nur der Vollständigkeit halber angegeben, da ein Datentyp Stack in Lisp keinen großen Nutzen hat, weil mit den Listenoperatoren car und cdr jede Liste direkt wie ein Stack behandelt werden kann.

Lisp ist durch das Fehlen von Namensräumen eher ungeeignet den Umgang mit ADTs zu lehren. Zudem haben sich funktionale Programmiersprachen in der Wirtschaft nur in Nischen durchgesetzt (z.B. bei Computeralgebrasystemen). Jedoch soll dies nicht bedeuten, dass funktionales Programmieren in der Lehre keine Rolle spielen sollte. Die starke Orientierung von Lisp am funktionalen Programmierparadigma macht es zu einer idealen Sprache, die Vor- und Nachteile dieser Art zu programmieren, zu vermitteln. Zudem gibt es Dialekte von Lisp, welche viele Unzulänglichkeiten (insb. das Fehlen von Namensräumen) des ursprünglichen Lisp beheben.

```
1  (define (stack_create)
2    nil
3  )
4
5  (define (stack_push stack elem)
6    (if (list? stack)
7      (cons elem stack)
8    else
9      'NOT_A_STACK
10   )
11 )
12
13 (define (stack_pop stack)
14   (if (empty? stack)
15     'EMPTY_STACK
16   else
17     (cdr stack)
18   )
19 )
20
21 (define (top stack)
22   (if (empty? stack)
23     'EMPTY_STACK
24   else
25     (car stack)
26   )
27 )
28
29 (define (empty? stack)
```

[1] Die Pragmatik hat auch vor Lisp nicht halt gemacht und daher gibt es die Befehle setcar und setcdr, welche den Zustand einer Liste doch verändern können.

[2] t.Lisp ist jedoch, anders als viele weit verbreitete Skriptsprachen (z.B. perl und php), stark typisiert.

```
30    (if (list? stack)
31      (nil? stack)
32    else
33      'NOT_A_STACK
34    )
35  )
```

Quellcode 2.8: Implementierung eines Stacks in t.Lisp.

Mithilfe der Operatoren setcar und setcdr wäre es auch in Lisp möglich einen zustandsverändernden Stack zu entwickeln. Dies wäre aber eine unnatürliche Vorgehensweise in einer so stark funktionalen Programmiersprache wie Lisp.

Quellcode 2.9 zeigt den Typ Sortierer welcher sich elegant in t.Lisp umsetzen lässt. Dies liegt vor allem daran, dass dieser Typ keine Zustände verwalten muss.

```
1   (define (>= a b) (or (> a b) (= a b) ) )
2   (define (join a b)
3   ; Verkettet zwei Listen: a.b
4     (cond
5       (nil? a) b
6       else
7         (cons (car a) (join (cdr a) b) )
8     )
9   )
10  (define (select op p l)
11  ; Selektiert alle Elemente in l, welche dem binären Prädikat op genügen.
12    (cond
13      (nil? l) l
14      (op (car l) p )
15        (cons (car l) (select op p (cdr l) ) )
16      else
17        (select op p (cdr l) )
18    )
19  )
20  (define (qsort l)
21  ; Erzeugt eine neue sortierte Version von l
22    (cond
23      (nil? l) l ; Die leere Liste ist bereits sortiert
24      (nil? (cdr l) ) l ; Die einelementige Liste bereits sortiert
25      else
26        ; Links < (car l) <= Rechts
27        (join
28          (qsort (select < (car l) l ) )
29          (cons (car l) (qsort (select >= (car l) (cdr l) ) ) )
30        )
31    )
32  )
33  (qsort '(1184 6801 4129 4797 1974 429) )
```

Quellcode 2.9: Implementierung von QuickSort in t.Lisp.

3. TeaJay

MB
MK

In diesem Kapitel wird der Entwurfsprozess von TeaJay dokumentiert und gleichzeitig die dabei entstandene Sprache beschrieben. Dabei werden, mit besonderem Fokus auf die Lehre, immer wieder Auswirkungen von Entwurfsentscheidungen und wie diese zu den Entwurfszielen stehen, diskutiert. Dabei ist der Entwurf von Programmiersprachen ein subjektives Thema und hängt auch stark von den gesetzten Zielen ab. Eine objektive Bewertung, ob eine Sprache „gut“ oder „schlecht“ ist, erscheint praktisch unmöglich. Dies beginnt schon bei der Diskussion, ob eine Sprache dem Nutzer möglichst viele Freiheiten lassen sollte und es damit erlaubt, denselben Sachverhalt auf viele unterschiedliche Weisen zu formulieren, oder ob eine Sprache versuchen sollte, dem Nutzer gewisse Muster aufzuzwingen, um ihn dadurch in seinen Formulierungen zu disziplinieren. Die Autoren versuchen, für TeaJay einen Mittelweg zu gehen. Alle für TeaJay getroffenen Entwurfsentscheidungen sind das Resultat einer Diskussion und Konsensbildung zwischen den Autoren dieser Arbeit. An besonders kontrovers diskutierten Punkten wird versucht, dem Leser einen Einblick in diese Diskussion zu geben.

3.1. Grundlegender Entwurf

Dieser Abschnitt soll einen Überblick über TeaJays grundlegenden Entwurf geben. Die folgenden Abschnitte dieses Kapitels gehen anschließend weiter ins Detail.

TeaJays Entwurf basiert vor allem auf dem ADT-Konzept und soll dieses in eine objektorientierte Umgebung einbetten. Um mit TeaJay nicht nur ein einziges Programmierparadigma lehren zu können, wird TeaJay als Multiparadigmensprache entworfen. Dabei ist TeaJay hauptsächlich objektorientiert, imperativ und strukturiert, unterstützt aber auch funktionales sowie generisches Programmieren. Das hat den Vorteil, dass es ohne einen Sprachwechsel möglich ist, die Grundzüge mehrerer Paradigmen zu lehren.

Die Entscheidung, TeaJay als hauptsächlich objektorientierte und imperative Sprache zu entwerfen, ist wohlüberlegt, denn zum einen *besitzt der imperative Stil einen besonderen Bezug zur Lebenswelt und zum Alltagsdenken der Schüler, und [zum anderen unterstützt er] insbesondere in seiner Erweiterung um objektorientierte Konzepte, die elementaren kognitiven Prozesse des Denkens, Erkennens und Problemlösens* [Sch95, S. 3]. Hinzu kommt, dass sich imperative Sprachen einer sehr weiten Verbreitung in Wirtschaft, Forschung und Lehre erfreuen.

Als Vorbild für TeaJay hat vor allem die Sprache Java gedient. Aus Sicht der Autoren bietet Java, als Vorbild für eine Lehrsprache, folgende Vorteile:

- Java ist eine, auch im wirtschaftlichen Umfeld, sehr weit verbreitete Programmiersprache und belegt auf dem *TIOBE-Index* (vgl. [TIO14]) Platz 2. Dabei

ist Java syntaktisch mit den ebenfalls weit verbreiteten Sprachen C (*TIOBE-Index* Platz 1) und C++ (*TIOBE-Index* Platz 4) verwandt.

- Java ist eine der beiden Sprachen, welche in den Vorgaben zu den unterrichtlichen Voraussetzungen für die schriftlichen Prüfungen im Abitur in der gymnasialen Oberstufe im Jahr 2014 [Zen14] und im Lehrplan für Informatik des Landes Hessen [Leh10] vorgeschlagen werden. Daher wird Java an vielen Schulen und Hochschulen seit Jahren als Lehrsprache eingesetzt und es gibt viele Anfänger mit Vorkenntnissen in dieser Sprache. Eine Orientierung von TeaJay an Java erleichtert ihnen einen Umstieg.
- Java ist eine Multiparadigmensprache und verfügt über eine automatische Speicherverwaltung.
- Java (Version 7) unterstützt, bis auf das Funktionale, alle Paradigmen, die auch TeaJay unterstützen soll[1].

TeaJay hat viele syntaktische Eigenschaften von Java übernommen oder erweitert. Daher dient die Java-Grammatik [GJS+12, Kapitel 18] als Ausgangspunkt für die Implementierung der TeaJay-Grammatik. Die größten Änderungen wurden gegenüber Javas Typsystem vorgenommen: TeaJay unterstützt keine Klassen, wie sie aus Java bekannt sind stattdessen Typen und Implementierungen. Ein TeaJay-Typ verhält sich dabei ähnlich wie eine Java-Klasse, enthält aber allein die Definition seiner Schnittstelle: Diese beinhaltet Instanzmethoden, statische Methoden und Konstruktoren, enthält aber keine Implementierungsdetails. Damit setzt TeaJay das Konzept des abstrakten Datentyps sehr kompromisslos um. TeaJay ermöglicht es Instanzen von Typen zu erzeugen, ohne dass dem Klienten eine Implementierung bekannt sein muss. In der Tat ist es in TeaJay nicht möglich, Implementierungen direkt zu instanziieren (mit einer Ausnahme, siehe dazu Abschnitt 3.4.3.8). TeaJays Typsystem weist weitere Besonderheiten auf:

- Es gibt keine primitiven Typen (wie z.B. int in Java). In TeaJay gibt es nur eine Klasse von Typen, nämlich Referenztypen.
- Es gibt keinen klassischen Arraytyp.
- TeaJay unterstützt Operatorüberladung.
- TeaJay kennt nur einen Typ zur Darstellung von Zahlen. Dieser Typ kann theoretisch alle rationalen Zahlen darstellen. In der Praxis ist die Genauigkeit durch den verfügbaren bzw. adressierbaren Speicherplatz beschränkt.

Durch die o.g. Entwurfsentscheidungen soll der sichtbare Unterschied zwischen eingebauten und benutzerdefinierten Typen möglichst gering ausfallen. Die Entscheidung, nur einen einzigen Typ zur Darstellung von Zahlen in die Sprache einzubauen, soll vor allem Anfängern zugutekommen. Weiterhin ist TeaJays Typsystem generisch und die meisten auf TeaJay anwendbaren Ausprägungen von Javas generischem Typsystem wurden übernommen. Zudem gibt es in TeaJay keine unsicheren automatischen

[1] Es ist in Java auch möglich funktional zu programmieren, allerdings auf eine eher indirekte Art.

Typkonvertierungen. Bedingt durch das Typsystem gibt es nicht einmal Bedarf für solche.

Da TeaJay sich bereits semantisch und syntaktisch an Java orientiert, liegt es nahe TeaJay, genauso wie Java, zur Java Virtual Machine [LYBB12], kurz JVM, zu übersetzen. Die JVM bietet für dieses Vorhaben auch einige Vorteile: So ist es auf ihr leicht möglich durch die Instruktion invokedynamic und die Reflection-API, Programmcode zur Laufzeit zu laden und dynamisch zu binden. Da ihr generische sowie parametrisierte Typen unbekannt sind, hat die Entscheidung, die JVM zu nutzen, dazu geführt, dass TeaJays generisches Typsystem, genauso wie das von Java, Typlöschung (siehe Abschnitt 3.4.3.7) durchführen muss. Dies ist ein rein technisches Eingeständnis und keine bewusste Entwurfsentscheidung. Die JVM bietet aber auch den Vorteil, sich um eine automatische Speicherbereinigung zu kümmern. Zudem bietet die Übersetzung zur JVM auch den Vorteil, dass TeaJay-Programme plattformunabhängig sind und potenziell auf allen Systemen, für die eine JVM (in der Version 7 oder höher) existiert, ausführbar sind. Für die Lehre ist dies vorteilhaft, da es wahrscheinlich ist, dass Lernende TeaJay auf ihrem gewohnten System nutzen wollen.

Durch die starke Trennung von Schnittstelle und Implementierung soll TeaJay das Programmieren zur Schnittstelle fördern, nach dem Prinzip *[p]rogrammieren auf eine Schnittstelle hin, nicht auf eine Implementierung* [EG96, S. 24]. Es ist in verschiedenen Sprachen möglich, mithilfe von Bibliotheken und Reflexion eine ähnliche Trennung von Schnittstellendefinition und Implementierung zu erreichen. Dies ist jedoch immer umständlich und erfordert daher ein hohes Maß an Disziplin. TeaJay vereinfacht diese Art zu programmieren durch seine Sprachmittel erheblich.

TeaJays Typsystem bietet, neben imperativen bzw. objektorientierten Sprachelementen, auch ein funktionales Sprachelement, die Closures (siehe Abschnitt 3.4.5). Diese erleichtern das Schreiben von funktionalen Programmen, haben aber auch eine pragmatische Seite: In TeaJay ist es nicht möglich, anonyme Typimplementierungen zu erstellen, da dies die starke Trennung von Typ und Implementierung untergraben würde. Closures stellen eine einfache Möglichkeit dar, Funktionalität zu kapseln und weiterzureichen, um z.B. auf einen Callback zu reagieren.

Genauso wie Java unterstützt TeaJay keine Mehrfachvererbung, sondern übernimmt das interface-Konzept von Java. Interfaces stellen in TeaJay eine partielle Typdefinition dar und sollen dazu dienen, sehr generelle Funktionalität aus Typen auszukoppeln (wie es z.B. Iterable<T> in Java tut).

Die Frage, ob TeaJay Operatorüberladung unterstützen soll, führte in der Entwurfsphase zu einer kontroversen Diskussion zwischen den Autoren. Sie führte schließlich zu dem in Abschnitt 3.4.6 vorgestellten Konzept der Operatorüberladung für TeaJay. Das Hauptargument für diese wurde durch das Entwurfsziel, eingebaute und benutzerdefinierte Typen möglichst gleichberechtigt zu behandeln, vorgegeben.

Um der starken Orientierung von TeaJay am Typbegriff gerecht zu werden, wurde in TeaJay ein Sprachelement namens typeswitch (siehe Abschnitt 3.4.7.5) eingeführt. Dieses vereinfacht nicht nur die Laufzeitprüfung von Typen, sondern macht den Umgang mit ihnen auch sicherer, da es hilft ungeprüfte explizite Typumwandlungen (englisch: *type casts*) zu vermeiden. Dennoch unterstützt TeaJay type casts, da typeswitch nur begrenzt mit parametrisierten Typen umgehen kann.

3.2. Hallo Welt in TeaJay

MB

Zur Einführung wird das klassische Beispielprogramm HalloWelt in TeaJay präsentiert. Ein ausführbares TeaJay Programm besteht mindestens aus einer Typdefinition und einer Implementierung, welche die Typdefinition realisiert. Die Typdefintion, beschreibt die Schnittstelle, die ein Typ besitzt.

Für den Typ HalloWeltProgramm ist zunächst ein Ordner hallowelt und in dieser eine Datei mit dem Namen HalloWeltProgramm.tj anzulegen. Ein Typ ist immer Mitglied eines Pakets und dies lässt sich über das Schlüsselwort package festlegen:

```
1 package hallowelt;
2 ...
```

Quellcode 3.1: HalloWeltProgrammImpl.tj

In TeaJay wird die Beschreibung der Schnittstelle eines Typs über das Schlüsselwort typedef eingeleitet, dabei ist es optional möglich, diesen Typ als öffentlich zu kennzeichnen:

```
1 package hallowelt;
2 /*
3  * Definiert ein Hallo Welt Programm
4  */
5 public typedef HalloWeltProgram {
6 ...
7 }
```

Quellcode 3.2: HalloWeltProgrammImpl.tj

Es ist zu beachten, dass der Name der Datei und der Typdefinition identisch sein müssen. Da der Typ ausgeführt werden soll, benötigt er eine Konstruktor- und eine Methodendefinition ohne Parameter. Die Methodendefinition darf dabei einen beliebigen Rückgabetyp besitzen. Dabei wird das Verhalten der Methode über einen Kommentar spezifiziert:

```
1  package hallowelt;
2  /*
3   * Definiert ein Hallo Welt Programm.
4   */
5  public typedef HalloWeltProgramm {
6    /*
7     * Akzeptiert keine Kommandozeilenparamter.
8     */
9    HalloWeltProgramm();
10   /*
11    * Gibt den String "Hallo Welt" auf der Konsole aus.
12    */
13   void run();
14 }
```

Quellcode 3.3: HalloWeltProgramm.tj

Es kann jetzt der Versuch unternommen werden, den Typ auszuführen. Mit dem TeaJay Compiler tjc wird zunächst die Typdefinition kompiliert:

```
1 $tjc hallowelt/HalloWeltProgramm.tj
```

Im Anschluss kann versucht werden, den Typ mit dem TeaJay-Programmstarter tj auszuführen.

```
1 $tj halloworld.HalloWeltProgramm run
2 Please state an implementation for hallowelt.↩
    ↪HalloWeltProgramm:
```

Dabei erwartet tj den Namen eines Typs, eine Methode und optional Kommandozeilenparameter zu erhalten. Da es noch keine Implementierung für den Typ gibt, fordert der Programmstarter den Benutzer auf, ihm anzugeben, wie eine konkrete Implementierung heißt. Für die Implementierung des Typs wird eine neue Datei angelegt und HalloWeltProgrammImpl.tj genannt. Damit eine sinnvolle Aufteilung von Typdefinition und Implementierung in TeaJay möglich ist, kann die Implementierung einem anderen Paket angehören. Die Implementierung des Typs ist Mitglied des Pakets implementations:

```
1 package implementations;
2 ...
```

Quellcode 3.4: HalloWeltProgrammImpl.tj

Die Datei befindet sich im Ordner implementations und dieser befindet sich im gleichen Verzeichnis wie der hallowelt-Ordner. Dadurch, dass sich die Implementierung in einem anderen Paket als der Typ, den sie implementiert, befindet, muss der Typ der Implementierung bekannt gegeben werden. Dies muss über die import-Anweisung geschehen oder alternativ kann der vollqualifizierte Name des Typs verwendet werden:

```
1 package implementations;
2 import hallowelt.HalloWeltProgramm;
3 ...
```

Quellcode 3.5: HalloWeltProgrammImpl.tj

Eine Implementierung wird gekennzeichnet über das Schlüsselwort typeimpl gefolgt von seinen Namen. Danach muss mit dem Schlüsselwort implements angegeben werden, welchen Typ die Implementierung implementiert:

```
1 package implementations;
2 import hallowelt.HalloWeltProgramm;
3 /* Stellt eine Implementierung des Typs hallowelt.Halloweltprogramm bereit */
4 typeimpl HalloWeltProgrammImpl implements HalloWeltProgramm {
5 ...
6 }
```

Quellcode 3.6: HalloWeltProgrammImpl.tj

Die Schnittstelle von HalloWeltProgramm definiert einen Konstruktor und die Methode run. Diese Schnittstelle muss die Implementierung implementieren. Der Konstruktor kann leer implementiert werden, da keine Initialisierungen durchgeführt werden:

```
1 package implementations;
2 import hallowelt.HalloWeltProgramm;
3 /* Stellt eine Implementierung des Typs hallowelt.Halloweltprogramm bereit */
4 typeimpl HalloWeltProgrammImpl implements HalloWeltProgramm {
5   HalloWeltProgrammImpl(){
6   }
7   ...
8 }
```

Quellcode 3.7: HalloWeltProgrammImpl.tj

Die Methode run ist für die Ausgabe von Hallo Welt verantwortlich. Hierfür wird die Methode IO.println(Object) aufgerufen. Dieser ist standardmäßig bekannt und muss daher nicht importiert oder über seinen vollqualifzierten Namen genutzt werden.

```
1 package implementations;
2 import hallowelt.HalloWeltProgramm;
3 /* Stellt eine Implementierung des Typs hallowelt.Halloweltprogramm bereit */
4 typeimpl HalloWeltProgrammImpl implements HalloWeltProgramm {
5   HalloWeltProgrammImpl(){
```

```
6   }
7   void run(){
8     IO.println("Hallo Welt");
9   }
10  }
```

Quellcode 3.8: HalloWeltProgrammImpl.tj

Wird die Implementierung wie folgt kompiliert,

```
1   $tjc implementations/HalloWeltProgrammImpl.tj
```

kann im Anschluss HalloWeltProgramm ausgeführt werden, wie nachfolgend gezeigt:

```
1   $tj hallowelt.HalloWeltProgramm run
2     Hallo Welt
```

3.3. TeaJays eingebaute Typen

MB
MK

Wie viele Programmiersprachen besitzt TeaJay eine kleine Zahl eingebauter Typen, welche immer vorhanden sind. Dabei unterstützen vor allem die Typen List und Number TeaJays High-Level-Charakter. TeaJay bietet folgende eingebaute Typen:

null : Der Typ null kann in TeaJay nicht ausgedrückt werden, da das Literal **null** immer für einen Wert vom Typ null steht. (vgl. 3.4.1.8)

void : Der Typ void steht für die Abwesenheit eines Rückgabewerts und darf nur als Rückgabetyp von Methoden vorkommen.

String : Der Typ String dient der Darstellung und Manipulation von Zeichenketten. Die interne Darstellung von String verwendet das Unicode-Format (vgl. [uni14]).

Boolean : Der Typ Boolean wurde in TeaJay, nach dem Vorbild von Java, eingebaut, um Wahrheitswerte darzustellen. Andere Programmiersprachen wie z.B. C kennen keinen solchen Typ und nutzen zur Darstellung von Wahrheitswerten ihre Zahltypen. Dabei steht 0 für falsch und jede Zahl ungleich 0 für wahr. Dies stiftet nach Meinung der Autoren nur Verwirrung und ist eine vermeidbare Fehlerquelle.

Closure : Der Typ Closure dient in TeaJay der Darstellung von Funktionsabschlüssen. Er sticht aus den eingebauten Typen besonders hervor, da er eine spezielle Syntax für Typparameter hat. Die Typparameter stehen dabei für den Rückgabetyp und die Parametertypen sowie deren Anzahl. (vgl. 3.4.5)

List : Der Typ List dient der Aggregation verwandter Objekte und hat, anders als ein Array, keine statische Größe sondern wächst nach Bedarf. Dabei lässt sich List zum einen ähnlich wie ein Array, über einen 0-basierten Index, nutzen und zum anderen wie eine Lisp-Liste (siehe 4.1.4.1). List soll dabei dem Sprachnutzer, zusammen mit der automatischen Speicherverwaltung, die Organisation von Speicher erleichtern und abstrahiert stärker als ein Array von der Hardware-Ebene. List hat unter den eingebauten Typen eine Sonderstellung: Seine Implementierung lässt sich austauschen und es ist möglich, diese vollständig in

TeaJay zu formulieren. Der Typ List wird trotzdem unter den eingebauten Typen aufgelistet, da er immer verfügbar ist und seine Typdefinition in Java verfasst wurde. Zudem wird List benötigt, um elementare Spracheigenschaften umzusetzen, wie z.B. Methoden mit variabler Argumentzahl. Auf den Typ List kann also nicht verzichtet werden.

Number : Laut Thomas E. Kurtz, einem der beiden Entwickler von BASIC, ist *[e]ines der schwierigsten Konzepte für einen Einsteiger [...] die Unterscheidung zwischen Integer- und Gleitkommazahlen* [BW09, S. 82]. TeaJay bietet daher nur einen Typ zur Darstellung von Zahlen. Diesen Ansatz haben bereits andere Lehrsprachen, wie z.B. BASIC oder die t.Sprachen (vgl. [Cla11, S. 2]), verfolgt. Dabei werden laut Kurtz in BASIC alle Zahlen als Gleitkommazahlen dargestellt (vgl. [BW09, S. 82]). Das kann bereits bei kleinen Zahlen zu einer ungenauen Darstellung führen. TeaJays Number-Typ stellt daher alle Zahlen als rationale Zahlen mit ganzzahligem Nenner und Zähler dar. Die Basisoperatoren arbeiten dabei exakt. Number kann potenziell jede rationale Zahl darstellen. Die tatsächliche Größe des darstellbaren Zahlraums wird jedoch immer durch eine Implementierung bzw. den verfügbaren Speicher der zugrundeliegenden Maschine begrenzt.

In Anhang B findet sich eine Dokumentation der TeaJay-Standardbibliothek. Dort werden auch die Schnittstellen aller Typen in TeaJay-Code angegeben.

3.4. Sprachelemente

Als Vorbild für die Aufteilung und Überschriften dieses Abschnitts hat die JLS gedient (vgl. [GJS+12]).

3.4.1. Lexikographische Struktur

MK

Die lexikographische Struktur von TeaJay lehnt sich stark an die von Java an. Java und TeaJay unterscheiden sich in diesem Aspekt hauptsächlich in der Auswahl der Schlüsselwörter und den zur Verfügung stehenden Operatoren. TeaJay unterstützt genau alle Formen von Literalen, die auch Java unterstützt, wobei diese in TeaJay meistens einen anderen Typ haben.

3.4.1.1. Schlüsselwörter

Folgende Wörter sind Schlüsselwörter in TeaJay:

abstract	assert	break	case	catch
continue	default	do	else	enum
extends	final	interfaces	for	if
implements	import	interface	native	new
package	private	public	return	static
super	synchronized	this	throw	throws
transient	try	void	volatile	while
typeswitch	typedef	typeimpl	closure	Closure

Einige der Schlüsselworte sind in TeaJay noch nicht einsetzbar und dienen als Platzhalter für zukünftige Funktionalität. Aus technischen Gründen kennt TeaJay momentan weitere Schlüsselwörter. Diese sind von Java geerbt und stellen sicher, dass es keine Konflikte mit der JVM gibt:

int float double short char
boolean byte strictfp class protected
long

3.4.1.2. Operatoren

TeaJays Operatoren sind aufgeteilt in überladbare und nicht überladbare Operatoren.

Überladbare Operatoren:

\+ * - / < > <= >= == != += *= -= /= [ ]

Nicht überladbare Operatoren:

&& || ! =

3.4.1.3. Separatoren

Folgende Symbole haben in TeaJay eine spezielle Bedeutung:

( ) { } ; , . : ... @ ?

3.4.1.4. Zahlen

TeaJay unterstützt alle Formen der in der Java Language Specification angegebenen Zahlliterale, siehe [GJS+12, Paragraph 3.10.1 - 3.10.2]. Diese haben in TeaJay jedoch alle den Typ teajay.lang.Number und keine Größenbeschränkungen. Hier seien nur einige Beispiele angegeben:

```
1  Ganze Zahlen:
2  0b111    // = 7
3  043L     // = 35
4  23       // = 23
5  0xFA     // = 250
6  Fließkommazahlen:
7  1.01
8  1.01d    // = 1.01
9  1.01f    // = 1.01
10 0.5e3    // = 500
11 0xf.3p2  // = 60.75
```

3.4.1.5. Zeichenketten

Zeichenketten werden in TeaJay durch " eingeleitet und durch ein weiteres " ausgeleitet (z.B. "hallo"). Das Zeichen \ hat innerhalb von Zeichenketten eine besondere Bedeutung: Es leitet einen Unicode-Escape, Oktal-Escape oder symbolischen Escape ein. Der Oktal-Escape hat folgende Syntax:

\(OctalDigit | OctalDigit OctalDigit | (0 | 1 | 2 | 3) OctalDigit OctalDigit)

Hier steht OctalDigit für alle Ziffern von Null bis Sieben. Beispiel:

```
\11 = \t (Tabulatorzeichen)
```

Der Unicode-Escape[2] wird mit \u eingeleitet und wird von folgender Regel produziert:

\u{u} HexDigit HexDigit HexDigit HexDigit

HexDigit steht hier für eine hexadezimale Ziffer. Diese Form des Escapes kann auch außerhalb von Zeichenketten im ganzen Quellcode verwendet werden. Beispiel:

```
\u0040 = @
```

Weitere symbolische Escapesequenzen sind:

\n | \r | \t | \b | \f | \\ | \" | \'

Zeichenketten haben in TeaJay immer den Typ String. Für weitere Informationen siehe [GJS+12, Paragraph 3.10.5-6].

3.4.1.6. Buchstabenliterale

Für Buchstabenliterale gelten die gleichen Escaperegeln wie für Zeichenketten. Buchstabenliterale werden in TeaJay durch ' eingeleitet und auch ausgeleitet. Ein Buchstabenliteral darf nur einen Buchstaben enthalten (nach Verarbeitung aller Escapes) und hat in TeaJay den Typ Number. Der Wert eines Buchstabenliterals wird immer eine ganze Zahl zwischen 0 und 65535 sein. Auch hier sei wieder auf die Java Language Specification verwiesen [GJS+12, Paragraph 3.10.4].

3.4.1.7. Wahrheitswerte

In TeaJay sind die beiden Wörter true und false mit einer besonderen Bedeutung versehen. Sie können daher nicht als Bezeichner dienen und haben immer den Typ Boolean.

3.4.1.8. Das null-Literal

Das null-Literal steht in TeaJay für die undefinierte Referenz bzw. die Abwesenheit eines Wertes. Sein Typ ist null, er ist Untertyp jedes anderen Typs und damit zuweisungskompatibel zu jedem TeaJay-Typ.

3.4.1.9. Bezeichner

TeaJay hält sich bei Bezeichnern an die in Java Language Specification[3]. Ein Bezeichner in TeaJay sieht wie folgt aus:

JavaLetter {JavaLetter | Digit}

[2] siehe auch [GJS+12, Paragraph 3.3]
[3] mit Ausnahme der Schlüsselwörter

Hier ist JavaLetter wie in [GJS+12, Paragraph 3.8] definiert und Digit steht für eine beliebige Dezimalziffer. Eine Ausnahme bilden alle Schlüsselwörter und die drei Literale **true**, **false** und **null**, welche nicht als Bezeichner vorkommen dürfen.

3.4.1.10. Kommentare

In TeaJay gibt es zwei Arten Kommentare auszudrücken[4]:

Zeilenübergreifend: Alles was zwischen /* und */ steht wird als Kommentar gewertet. Das gilt insbesondere auch für Zeilenumbrüche. Beispiel: /* beliebiger Text */

Mit dem Zeilenende endend: Der Rest der Zeile wird nach // als Kommentar gewertet. Beispiel: //beliebiger Text außer Zeilenumbruch

3.4.2. Pakete

MB

Eine Herausforderung beim Programmieren ist der Umgang mit Namenskonflikten von Entitäten, die beispielsweise durch die Verwendung von Bibliotheken oder durch Gruppenarbeit entstehen können. Um diese Namenskonflikte zu vermeiden, haben viele Programmiersprachen ein Konzept, das es erlaubt, ein Programm in Module zu unterteilen. In TeaJay wird dies durch die Organisation in Pakete geleistet. Hier wird durch folgende Regel bekannt gegeben, dass die nachfolgende Entität Mitglied des Pakets ist:

“package“ <Identifier> { “.“ <Identifier> }

In TeaJay ist es erlaubt, pro Datei genau eine Entität zu beschreiben und diese darf nur Mitglied eines Pakets sein.

Die Paketorganisation von TeaJay erlaubt es, Entitäten so zu deklarieren, dass sie nur im Paket sichtbar sind und mit keinem TeaJay-Mechanismus nach außen bekannt gegeben werden können. Durch die Steuerung der Sichtbarkeit kann auch beim Paketkonzept von Schnittstellen und Implementierungen gesprochen werden. Nach außen sichtbare Entitäten bilden die öffentliche Schnittstelle des Pakets, während der Rest nur Implementierungsdetail ist. Für einen Lehrenden ist es entscheidend, dem Lernenden diesen Umstand begreifbar zu machen, da das Nachdenken über Schnittstellen schon mit der Sichtbarkeit eines Typs beginnt. Eine weitere Eigenschaft der Paketorganisation ist, dass sich Entitäten, die Mitglieder im gleichen Paket, sich gegenseitig kennen. Das bedeutet, innerhalb eines Pakets muss eine Entität nicht importiert werden.

Die Strategie, um potenzielle Namenskonflikte in TeaJay zu vermeiden ist, dass sich der echte Name, also der vollqualifizierte Name einer Entität, aus Paketbezeichnung und Benennung der Entität zusammensetzt.

Um die öffentlichen Entitäten eines Pakets außerhalb dieses Pakets zu nutzen, gibt es zwei Möglichkeiten: Zugriff über den vollqualifizierten Namen oder die Verwendung des Schlüsselwortes import. Die Regel

[4] siehe auch [GJS+12, Paragraph 3.7]

“import“ <Identifier> { “.“ <Identifier> } [“.*“]

beschreibt die Importanweisung. Mit ihr wird die Benennung einer Entität bekannt gegeben. Danach muss der vollqualifizierte Name nicht verwendet werden. Wenn mehrere Importanweisungen die gleichen Namen von Entitäten laden, wird die Benennung der letzten Entität, die über import bekannt gegeben wurde, genutzt. Durch das Wildcard Symbol .* kann ein Importieren aller öffentlichen Entitäten aus einem Paket durchgeführt werden.

3.4.3. Typen und Implementierungen

MB
MK

In TeaJay steht das Konzept des ADTs im Fokus des Entwurfs der Programmiersprache. Wie in Kapitel 2 beschrieben, ist ein ADT die Spezifikation der Schnittstelle eines Typs. Diese ist von seiner konkreten Implementierung getrennt. Des Weiteren darf in TeaJay eine kompilierbare Datei genau eine Typdefinition, eine Typimplementierung, einen Aufzählungstyp oder eine Interfacedefinition besitzen. Diese Umsetzung erlaubt es einem Entwickler, sich auf eine Aufgabe zu konzentrieren. Der Aufbau einer kompilierbaren Datei ist in folgender Regel dargestellt:

[<Package>] { <Import> } <TypeDecl>
<TypeDecl> ::= <Typedef> |<Typeimpl> |<Enumdecl> |<Interfacedecl>

3.4.3.1. Typdefinitionen

MB

Die Entwicklung eines abstrakten Datentyps in TeaJay beginnt mit der Definition seiner Schnittstelle. Hierbei muss der Entwickler folgende Fragen bei der Definition eines Typs beantworten:

1. Ist der Typ nur im Paket sichtbar oder soll er auch außerhalb des Pakets genutzt werden?
2. Kann vom Typ geerbt werden oder gibt es keine sinnvolle Spezialisierung von ihm?
3. Ist seine Schnittstelle unabhängig von einem konkreten Typ, so dass diese für viele Typen genutzt werden kann?
4. Ist der Typ eine Spezialisierung eines anderen Typs?

Die Antworten auf diese Fragen kann der Entwickler durch folgende Regel in TeaJay beschreiben:

[“public“] [“final“] “typedef“ <Identifier> [<Generic>] [<Derivation>] “{“
<MethodOrConstrcutorDecl> [<ThrowsList>] “;“ “}“

Der Zugriffsmodifizierer public dient dazu, die Sichtbarkeit der Typdefinition zu steuern. Das nicht-Anführen von public führt dazu, dass der Typ nur innerhalb des gleichen Pakets genutzt werden kann. Im Gegensatz dazu ermöglicht das Anführen, dass der Typ auch außerhalb des Pakets genutzt werden kann. Das Setzen des Schlüsselworts final bei der Typdefinition sagt aus, dass von diesem Typ nicht weiter geerbt

werden kann. Der Bezeichner in der Regel ist der Name des Typs. Hierbei wird empfohlen, dass der Name die Aufgabe des Typs widerspiegelt.

In TeaJay gliedert sich die Schnittstelle eines Typs in Konstruktor- und Methodendefinitionen. Das Verhalten wird in TeaJay durch Kommentare spezifiziert. Das bedeutet, der Entwickler hat die Freiheit die Aufgaben und das Verhalten eines Typs in natürlicher Sprache zu beschreiben. Es gab die Überlegung, die Kommentierung des Typs durch den Compiler durchzusetzen. Dies wurde nicht weiter verfolgt, da der Compiler keine Prüfung der Sinnhaftigkeit der Kommentare durchführen kann und die Entwickler im schlimmsten Fall schlechte Kommentare formulieren. Hinzu kommt, dass einige Methoden durch die geeignete Wahl des Namens eine korrekte Spezifizierung ihrer Aufgabe und Verhaltens ergeben und dadurch ein Kommentar überflüssige Redundanzen erzeugt. Eine Konstruktordefinition wird durch diese Regel beschrieben:

<Identifier> “(“ [<Parameters>] “)“

Die Konstruktordefinitionen der Schnittstelle spezifizieren, was für die Initialisierung eines Typs benötigt wird, damit bei der Initialisierung einer Instanz des Typs dieser in einem korrekten Anfangszustand überführt wird. Weiterhin muss der Bezeichner des Konstruktors mit dem Bezeichner des Typs identisch sein. Für ausführbare Typen, die über tj gestartet werden können, gilt, dass es mindestens einen Konstruktor gibt, der entweder keinen, nur Strings oder List<String> als Parameter erwartet. Die Methodendefinitionen können Instanz- oder Typmethoden (statisch) sein. Instanzmethoden werden durch diese Regel beschrieben:

<Type> <Identifier> “(“ [<Parameters>] “)“

Instanzmethoden beschreiben, den Austausch von Nachrichten zwischen Instanzen. Für ausführbare Typen gilt, es muss eine Methodendefinition geben, die keine Parameter besitzt. Die Typmethoden werden in TeaJay so beschrieben:

“static“ [<Generic>] <Type> <Identifier> “(“ [<Parameters>] “)“

Diese Methoden gehören keiner Instanz eines Typs an. Werden sie bei einer Typdefinition angegeben, so sind sie das Äquivalent zu einer globalen Funktion. Des Weiteren werden statische Methoden nicht vererbt. In TeaJay können nur statische Methoden generisch parametrisierbar sein, dies ist aber nur ein technischer Aspekt.

Bei der Definition von Methoden und Konstruktoren ist es möglich, sie mehrfach mit gleichen Namen zu definieren. Sie werden anhand ihrer Parameter unterschieden. Der Rückgabetyp wird dabei nicht mit betrachtet, da es ansonsten zu Problemen beim Aufruf von Methoden kommt, wie im nachfolgenden Beispiel gezeigt:

```
1  Number func(){
2    ...
3  }
4  Boolean func(){
5    ...
6  }
7  void test(){
8    /* Ermittlung der korrekten
```

```
9       func-Methode möglich. */
10      Number a = func();
11      /* Hier ist die Ermittlung
12      nicht möglich */
13      func();
14    }
```

Es kommt zu einem Kompilierfehler, wenn Methoden oder Konstruktoren den gleichen Namen und die gleichen Parameter besitzen. Ein Typ in TeaJay darf Operatoren überladen. Das Überladen geschieht über das Erstellen von Methoden mit festgelegten Namen und Parametern (siehe Kapitel 3.4.6).

Die Parameterdefinition einer Methode unterscheidet sich nicht von der in Java oder C, sie ist in TeaJay wie folgt definiert:

<Parameters>::= <Type> (<Identifier> [ParamRest] | "..." <Identifier>)
<ParamRest>::= "," Parameters

Methoden, die eine variable Parameteranzahl haben können, werden durch das Symbol **...** kenntlich gemacht. Festzuhalten ist, dass in Java Methoden mit variabler Parameterzahl auch einen Array übergeben bekommen können. Diese Fähigkeit ist mit dem Typ List allerdings noch nicht gegeben.

In TeaJay ist es möglich, optional anzugeben, dass eine Methode oder ein Konstruktor Ausnahmen werfen kann. Dies geschieht über die Angabe einer Ausnahmeliste, die so definiert ist:

<ThrowsList>::="throws" <Type> { "," <Type> }

Die Typen in der Ausnahmeliste müssen zuweisbar zu java.lang.Throwable sein, ansonsten kommt es zu einem Kompilierfehler.

3.4.3.2. Aufzählungstypen

In TeaJay gibt es die Möglichkeit Aufzählungstypen zu erstellen. Diese haben wie die Typen einen Bezeichner, im Gegensatz dazu allerdings keine Methodenschnittstelle, die der Entwickler definieren kann. Anders als die Aufzählungstypen in Java oder C, können Aufzählungstypen in TeaJay nur symbolische Konstanten definieren. Diesen können jedoch keine nutzerspezifischen Werte zugewiesen werden. Dies ist der Prototypeigenschaft des Compilers geschuldet. Die Aufzählungstypen sind nach folgender Regel definiert:

["public"] "enum" <Identifier> "{" [<Identifier> { "," <Identifier> }] "}"

Im diesem Beispiel wird ein Aufzählungstyp für Wochentage präsentiert:

```
1   public enum Wochentag {
2     Montag, Dienstag, Mittwoch, Donnerstag,
3     Freitag, Samstag, Sonntag
4   }
```

Nachdem ein Aufzählungstyp definiert wurde, lässt er sich wie jeder andere Typ in TeaJay nutzen. Jedoch gibt es einige Besonderheiten zu beachten. Eine Konstante definiert eine Instanz des Aufzählungstyps. Des Weiteren ist es nicht möglich eine

Instanz des Aufzählungstyps direkt zu erzeugen. Hierdurch können die Konstanten des Aufzählungstyps nur, wie im Beispiel gezeigt, zugewiesen werden: Wochentag↩ ↪ day = Wochentag.Montag. Der Aufzählungstyp lässt sich wie im folgenden Beispiel nutzen:

```
1  public void Terminplan(Wochentag day){
2    if(day == Wochentag.Montag){
3      IO.println("Arbeit von 8:00 - 19:00 Uhr");
4    }else {
5      IO.println("Urlaub: "+ day.toString());
6    }
7  }
```

Die Konstanten des Aufzählungstyps sind wie jeder Typ in TeaJay untereinander vergleichbar. Des Weiteren implementiert jeder Aufzählungstyp das Interface TJComparable und besitzt unter anderem eine Instanzmethode toString() und zwei statische Methoden valueOf() und getConstants()

toString():
Die Methode gibt den Namen der Konstanten zurück.

valueOf(String name):
Die Methode gibt die Konstante mit Namen name zurück.

getConstants():
Die Methode gibt eine Liste aller Elemente des Aufzählungstyps zurück.

In TeaJay wurde der Aufzählungstyp als Bestandteil der Sprache aufgenommen, da es Typen gibt, die durch eine kleine Anzahl von Konstanten repräsentiert werden können. Daher werden sie durch einen Aufzählungstyp besser dargestellt als durch eine Schnittstelle.

3.4.3.3. Generische Typen

MK

Es ist möglich, generische Typen wie folgt in TeaJay zu definieren:

["public"] "typedef" <Identifier> "<" <TypeVariable> {"," <TypeVariable>} ">"
["extends" <Type>]
<TypeVariable> ::= <Identifier> ["extends" <Type>]

Generische Typen lassen sich in TeaJay ähnlich wie in Java definieren. Dabei ist ein generischer Typ in TeaJay ein reiner Metatyp und es können nur parametrisierte Instanzen (siehe Abschnitt 3.4.3.4) von ihm genutzt werden. Damit steht ein generischer Typ auf Sprachebene für eine Menge von parametrisierten Typen. Das Format der Typvariablen (englisch: *type variables* [GJS+12, Paragraph 4.4]) lehnt sich ebenfalls an Java an. TeaJay unterstützt dabei lediglich keine Mehrfachangabe von Schnittstellengrenzen: Es ist also nicht möglich, zu fordern, dass eine Typvariable mehrere interfaces erfüllen soll. Für jede Typvariable kann also nur eine Typgrenze angegeben werden. Dabei kann es sich auch um ein interface handeln. Typgrenzen legen fest, mit welcher Art von Typen der generische Typ parametrisiert werden

darf: Ist eine Typgrenze angegeben, so können für die Typvariable nur Untertypen ihrer Typgrenze eingesetzt werden. Es folgt ein Beispiel:

```
1  typedef SortedSet<T extends TJComparable<T>> {
2    SortedSet();
3    void add(T elem);
4    TJIterator<T> iterator();
5  }
6  //Code:
7  SortedSet<Number> numberSet = new SortedSet<Number>(); //OK
8  SortedSet<TJObject> tjobjectSet = new SortedSet<TJObject>(); //Fehler
```

Die Parametrisierung SortedSet<Number> ist zulässig, da Number das interface TJComparable<Number> erfüllt. SortedSet<TJObject> ist dagegen unzulässig, da TJObject dieses nicht erfüllt.

3.4.3.4. Parametrisierte Typen

Parametrisierte Typen haben in TeaJay folgende Syntax:

<Type> ::= <QID> "<" <ParamType> {"," <ParamType>} ">"
<QID> ::= <Identifier> { "." <Identifier> }
<ParamType> ::= <QID> | "?" | "?" ("extends" | "super") <Type>

Beispiele parametrisierter Typen:

```
1  List<String> , List<? extends TJIterable<?>> , TJIterable<String>
```

In diesem Fall wurde der generische Typ List z.B. mit dem Typargument String parametrisiert und eine parametrisierte Instanz des generischen Typs List erzeugt. TeaJay unterstützt auch die aus Java bekannten Platzhalter (englisch: *wildcards*), in derselben Form wie in [GJS+12, Paragraph 4.5] angegeben.

Anders als Java, kennt TeaJay keine sogenannten *raw types* [GJS+12, Paragraph 4.8], was zur Folge hat, dass bei parametrisierten Typen immer Typargumente angegeben werden müssen. In Java ist es durch *raw types* möglich das generische Typsystem zu umgehen und unparametrisierte Instanzen von generischen Typen zu erzeugen bzw. Parametrisierungen von Typen, ohne explizite Typumwandlung, zu entfernen.

Das generische Typsystem von TeaJay ist für eine Lehrsprache sehr komplex. TeaJay ist aber eine Sprache, die das Denken über Typen in den Vordergrund stellen möchte und sollte daher, auch um fortgeschrittene Programmierer nicht zu frustrieren, ein mächtiges generisches Typsystem haben. Zudem hätte die Beantwortung der Frage, wie es möglich ist, ein ähnlich mächtiges generisches Typsystem, welches gleichzeitig für Anfänger einfacher zu verstehen ist, zu entwickeln, den Rahmen dieser Arbeit gesprengt. Es wäre auch ohne ein generisches Typsystem schwierig, den Typ List ohne eine große Sonderbehandlung durch das Typsystem zu implementieren. Daher wurde, im Zuge der Gleichbehandlung von eingebauten und benutzerdefinierten Typen, TeaJay mit einem generischen Typsystem ausgestattet.

3.4.3.5. Vererbung

MB

Ein wichtiges Konzept, welches viele objektorientierte Programmiersprachen besitzen, ist die Vererbung. Durch sie ist es möglich über Supertyp bzw. Basistyp und Subtyp nachzudenken. Meyer diskutiert und beschreibt verschiedene Arten von Vererbung (vgl. [Mey96]), die wichtigsten sind:

- **Erben der Schnittstelle und Implementierung:**
  Diese beschreibt eine ist-ein-Beziehung zwischen Typen. Das heißt, überall, wo der Basistyp erwartet wird, ist es möglich einen Subtyp einzusetzen. Des Weiteren wird die Funktionalität des Basistyps auf den Subtyp übertragen. Dieser kann falls nötig Teile der Funktionalität überschreiben.

- **Erben der Schnittstelle:**
  Diese beschreibt ebenfalls eine ist-ein-Beziehung zwischen Typen, allerdings wird nur die Schnittstelle ohne Funktionalität vererbt. In Java z.B. liegt diese Art der Vererbung vor, wenn eine Klasse ein Interface implementiert.

- **Erben der Implementierung:**
  Das ausschließliche Erben der Implementierung ist ein Konzept, um Funktionalität wiederzuverwenden. Durch diese Art des Erbens ist es möglich, schnell mächtige Typimplementierungen zu realisieren(vgl. [EG96]). Dieses Konzept ist in Java nicht realisiert aber in C++. Diese Vererbungsart wird in C++ durch die Fähigkeit des privaten Erbens erreicht. Eine Konsequenz dieses Vererbungsprinzips ist, dass der erbende Typ und der geerbte Typ in keiner ist-ein-Beziehung stehen.

In TeaJay existiert nur die ist-ein-Vererbung. Das bedeutet für den Entwickler, dass die Typdefinition die Schnittstelle ihres Basistyps zur Kompilierzeit erbt. Zur Laufzeit erbt dann eine Implementierung von ihrer Basistypimplementierung. Dabei steht es dem Implementierer des Subtyps frei, die geerbte Schnittstelle in Teilen oder ganz zu überschreiben. Des Weiteren kann ein Typ angeben, dass seine Implementierungen Interfaces implementieren.

Die Mehrfachvererbung in der Form, dass von mehreren Typdefinitionen zur Kompilierzeit und Implementierungen zur Laufzeit geerbt werden kann, ist in TeaJay nicht realisiert, da mit ihr Probleme verbunden sind, wie das Diamant-Problem (vgl. [TJJV04]) und es auch zu Realisierungsschwierigkeiten auf der JVM käme. Grundsätzlich ist es so, dass die Mehrfachvererbung, sinnvoll angewendet, nützlich für den Entwurf von Typen ist. Allerdings existiert in TeaJay die Möglichkeit, von mehreren Interfaces zu erben.

Ein Aspekt der Vererbung, der häufig in objektorientierten Sprachen existiert, ist die Möglichkeit, geschützte Methoden zu vererben. Diese besteht in TeaJay nicht, als eine direkte Konsequenz daraus, dass ein Typ vollständig durch seine öffentliche Schnittstelle beschrieben wird. Eine weitere Eigenschaft von TeaJay ist, dass es keine abstrakten Typen gibt. Dies sind Typen, die Teile ihrer Schnittstelle von Subtypen realisieren lassen. Dass es keine abstrakten Typen gibt, ist der Prototypeneigenschaft von TeaJay geschuldet.

Ein weiterer Aspekt, der bei der Implementierung von TeaJay diskutiert wurde, ist, ob der erbende Typ bei seiner Typdefinition die Schnittstelle des Basistyps mitangeben muss. Dies hätte bedeutet, dass Quelltext dupliziert würde und dies wahrscheinlich zu vielen Fehlern führen könnte. Zudem würden es den frustrations Faktor bei der Entwicklung mit TeaJay erhöhen.

In TeaJay erbt jeder benutzerdefinierte Typ direkt oder indirekt von TJObject und dadurch auch von java.lang.Object. Hierdurch besitzt jeder Typ folgende Schnittstelle:

```
typedef TJObject  {
    Number tjHashCode();
    Boolean operator_eq(Object obj);
    String toString();
}
```

Durch die Nutzung folgender Regel bei der Typdefinition wird die Vererbung realisiert:

<Derivation>::="extends" <Type> ["interfaces" <Type> {,<Type>}]

Die Regel extends <Type>, gibt den Typ an, von dem die Schnittstelle geerbt werden soll. Der letzte Teil der Regel gibt an, von welchen Interfaces ein Typ zusätzliche Schnittstellen erhalten soll.

Dadurch, dass jeder Typ in TeaJay direkt oder indirekt von Object erbt, ist es möglich, einen Ableitungsbaum zu konstruieren, der die Verwandtschaftsbeziehungen zwischen allen verwendeten Typen repräsentiert. Alle Knoten im Baum repräsentieren einen Typ bzw. ein Interface. Der Wurzelknoten repräsentiert dabei Object. Die Kinderknoten eines Knotens besitzen immer eine Kante, die zurück zu ihm führt. Dies symbolisiert die ist-ein-Beziehung zwischen den Typen. Der Ableitungsbaum wird detailliert in Kapitel 5.3.11 geklärt.

3.4.3.6. Zuweisungskompatibilität von Typen

MK

Die Definition der Zuweisungskompatibilität in TeaJay folgt der Java Language Specification [GJS+12, Paragraph 5.5], mit folgenden Ausnahmen:

- TeaJay kennt keine primitiven Typen und Arrays, mit einer Ausnahme: Der cast-Operator ist in der Lage zu Java-Arrays zu casten. Dabei sind auch Arrays von primitiven Java-Typen erlaubt, nicht jedoch primitive Typen selber. Weitere Informationen finden sich in Abschnitt 3.5.
- Im Sinne der Zuweisungskompatibilität erben Implementierungen von ihrer Typdefinition.
- Es gibt in TeaJay keine implizite, einengende Typkonvertierung[5].
- Im Falle von Closures ist die Zuweisungskompatibilität wie folgt definiert: Ein Closure vom Typ Closure<X>($Y_0, Y_1, \ldots, Y_n$)) ist genau dann zuweisungskompatibel zu einem Closure vom Typ Closure<A>($B_0, B_1, \ldots, B_n$)), wenn X ein Untertyp von A ist und für alle $i \in [0, n]$ gilt: Y_i ist ein Supertyp von B_i. Damit ist z.B. folgende Zuweisung gültig:

```
    Closure<Object>(String, Number) cl2 = closure<Boolean>(String a, Object b) {
      ...
    };
```

Ein Typ B ist immer dann zuweisungskompatibel zu einem Typ A, wenn A ein Supertyp von B ist. Im Falle unparametrisierter Typen kann die Supertyp-Beziehung

[5] engl. narrowing conversion, siehe auch [GJS+12, Paragraph 5.1.6]

einfach durch den Ableitungsbaum festgestellt werden. Dabei dürfen dann auch im Erbpfad keine parametrisierten Typen auftauchen. Zuweisungskompatibilität im Falle parametrisierter Typen ist für Anfänger nicht sehr intuitiv, denn hier ist die Frage, ob A ein Supertyp von B ist deutlich schwerer zu beantworten, wie folgendes Beispiel zeigt:

```
1  List<Object> objectList = null;
2  List<String> stringList = new List<String>();
3  //Auf den ersten Blick ist diese Zuweisung unkritisch:
4  objectList = stringList; //Fehler
```

List<String> ist nicht zuweisungskompatibel zu List<Object>, obwohl String Untertyp von Object ist. Durch diese Zuweisung würde aber die Eigenschaft von stringList, eine Liste von Strings zu sein, verloren gehen und es wird möglich Objekte anderen Typs der Liste hinzuzufügen. Über stringList könnte aber noch als List<String> auf die Liste zugegriffen werden, jedoch ist dann nicht mehr garantiert, dass jedes Element der Liste vom Typ String ist (damit wäre die statische Typsicherheit nicht mehr garantiert). Dieses Problem wird von den Wildcard-Typen adressiert, daher folgt hier ein kleiner Überblick über diese:

? extends <Type> :

```
1   typedef A extends TJObject { //extends TJObject kann auch weggelassen werden
2   }
3   typedef B extends A {
4   }
5   //Code:
6   List<B> blst = new List<B>();
7   blst.add(new B());
8
9   //? extends A gibt die Zusicherung, dass alle Elemente in lst zuweisungskompatibel ↩
        ↪zu A sind.
10  List<? extends A> lst = blst;
11  A a = lst.get(0); //OK
12  lst.set(0, new A()); //Fehler
13  lst.set(0, new B()); //Fehler
14  lst.set(0, null); //OK
```

Der Platzhalter ? extends A steht für einen beliebigen Untertyp von A. Damit ist ? extends A zu jedem Supertyp von A zuweisungskompatibel[6] (wie z.B. TJObject oder A selber). Mehr Informationen enthält der Platzhalter nicht, daher ist nur der Typ null zuweisungskompatibel zu einem Platzhalter der Form ? extends ....

? super <Type> :

```
1   List<TJObject> blst = new List<TJObject>();
2   blst.add(new B());
3
4   //? super A sorgt dafür, dass nur Untertypen von A der Liste hinzugefügt werden kö↩
        ↪nnen.
5   List<? super A> lst = blst;
6   A a = lst.get(0); //Fehler
7   Object obj = lst.get(0); //OK
8   lst.set(0, new A()); //OK
9   lst.set(0, new B()); //OK
10  lst.set(0, null); //OK
```

Der Platzhalter ? super A steht für einen beliebigen Supertyp von A. Daher ist jeder Untertyp von A zuweisungskompatibel zu ? super A. ? super ... ist im Gegenzug nur zu java.lang.Object[7] zuweisungskompatibel.

[6] siehe 3.4.3.6

[7] Aus Kompatibilitätsgründen ist auch in TeaJay java.lang.Object und nicht TJObject der allgemeinste Typ.

? :

```
1  List<String> stringList = new List<String>("a", "b");
2  List<?> anyList = stringList;
3
4  Object obj = anyList.get(0); //OK
5  anyList.add("hallo"); //Fehler
6  anyList.add(new Object()); //Fehler
7  anyList.add(null); //OK
```

Der Platzhalter ? steht für einen beliebigen Typ. Er ist daher nur zuweisungskompatibel zu java.lang.Object, während der einzige zu ? zuweisungskompatible Typ null ist.

Beinhalten X und Y keine Wildcard-Typen, so ist z.B. A<X> genau dann zuweisungskompatibel zu A<Y>, wenn X=Y gilt. Dabei stehen X und Y für Mengen von Typen. Sollte X ein Wildcard-Typ sein und Y nicht ($|Y| = 1$), so ist die Zuweisung A<X> a = new A<Y>(); erlaubt, falls $Y \subseteq X$ gilt.

```
1  List<? super Number> lst = new List<Object>();
2  List<? extends TJIterable<?>> ilst = new List<List<Number>>();
```

Auf eine vollständige Dokumentation des Regelwerks wird hier verzichtet und stattdessen auf die JLS (vgl. [GJS+12]) und [ora13] verwiesen.

3.4.3.7. Typlöschung

In TeaJay werden, wie in Java, die Typargumente von parametrisierten Typen beim Übersetzen gelöscht. Die Typsicherheit wird also zur Kompilierzeit überprüft und danach nur noch eingeschränkt zur Laufzeit[8]. Beispielsweise ist der Laufzeittyp von List<String> und List<Number> identisch. Dies sollte man im Hinterkopf haben, wenn man z.B. den Cast-Operator[9] nutzt. Zudem lassen sich, bedingt durch die Typlöschung, die beiden folgenden Methoden zur Laufzeit nicht unterscheiden:

```
1  typedef SomeType {
2    void test(List<String> strList);
3    void test(List<Number> numList);
4  }
```

Es ist tatsächlich nicht möglich diese beiden Methoden im selben Class-File unterzubringen. Der Compiler wird daher obige Typdefinition ablehnen.

Auch generische Typen erfahren eine Typlöschung. Dies hat aber aus Sicht des Sprachnutzers kaum Auswirkungen und wird eher zu den Implementierungsdetails gezählt. Weitere Informationen finden sich in [GJS+12, Paragraph 4.6 - 4.7]. Da TeaJay keine Arrays und verschachtelte Typen kennt, finden diese Regeln für TeaJay keine Anwendung.

3.4.3.8. Implementierungen

In TeaJay werden Implementierungen von Typen nach folgender Regel angegeben:

<Typeimpl> ::= "typeimpl" <Identifier> [<Generics>] "implements" <Type> <TypeImplBlock>

<TypeImplBlock> ::= { <FieldDeclaration> } ["static" <Block>] { <MethodOrConstructorImpl > }

[8] Der Bytecode der JVM ist statisch stark typisiert, d.h. jeder Typfehler wird früher oder später zu einer ClassCastException führen oder der Verifizierer wird die .class-Datei ablehnen.

[9] siehe 3.4.6.1

Implementierungen müssen immer den Namen der Typdefinition angeben, welche sie implementieren. Dabei muss die generische Signatur der Implementierung exakt mit der des implementierten Typs übereinstimmen[10]. Da die generischen Typinformationen zur Laufzeit nicht mehr vorliegen (vgl. Abschnitt 3.4.3.7), ist es nicht möglich, spezialisierte Implementierungen anzugeben. Soll z.B. ein Stack speziell für Zahlen (Number) implementiert werden, so ist folgendes Vorgehen nicht möglich:

```
typedef Stack<T extends TJObject> {
  ...
}
//ungültige Implementierung
typeimpl NumberStack implements Stack<Number> {
  ...
}
```

In Kapitel 6 werden Möglichkeiten diskutiert, spezialisierte Implementierungen umzusetzen. Eine gültige Implementierung von Stack<T> kann wie folgt aussehen:

```
typeimpl GenericStackImpl<T extends TJObject> implements Stack<T> {
  ...
}
```

Es ist notwendig, dass bei der Angabe des implementierten Typs die Typvariablen, wie im Beispiel zu sehen, wiederholt werden. Zudem müssen die Signaturen der öffentlichen Methoden exakt mit der Signatur der entsprechenden Methoden in der Typdefinition übereinstimmen (insb. auch die throws-Liste). Eine Implementierung muss alle Methoden, die in ihrer Typdefinition bzw. dessen interfaces vorkommen implementieren. Methoden, die von einem Supertyp geerbt werden, können überschrieben werden.

Konstruktoren tragen den Namen der Implementierung und alle Felddeklarationen müssen vor den Methoden- bzw. Konstruktordeklarationen stehen:

```
typedef MyType {
  MyType(); //MyType.<init>()
}
typeimpl MyTypeImpl implements MyType {
  Number index = 4;
  static String greeting;
  static { //static-Konstruktor
    greeting = "Bonjour";
  }
  //Konstruktor
  MyTypeImpl() { //Implementiert den Konstruktor MyType.<init>()
  }
  private MyTypeImpl(MyTypeImpl impl) { //Private Konstruktoren sind erlaubt
    //copy-constructor
  }
}
```

Der static-Konstruktor kann nicht explizit aufgerufen werden und wird nur ein einziges Mal beim Laden der Implementierung ausgeführt (bevor irgendeine andere Methode der Implementierung zur Ausführung kommt). Es darf nur einen static-Konstruktor pro Implementierung geben und er darf nicht mit einem expliziten return verlassen werden. Momentan werden statische Felder nur initialisiert, wenn ein static-Konstruktor angegeben wurde (dieser kann leer sein).

In TeaJay sind Implementierungen sehr eingeschränkte Typen: Sie sind außerhalb ihres eigenen Namensraums nicht sichtbar (jede Implementierung hat ihren eigenen privaten Namensraum). Klientencode kann nur über den Namen des implementierten Typs mit einer Implementierung kommunizieren und damit auch nur über die öffentliche Schnittstelle dieses Typs. Der Klient hat dadurch auch keine direkten

[10] aus technischen Gründen gilt das sogar für die Bezeichnung der Typvariablen

Informationen darüber, mit welcher Implementierung er gerade kommuniziert. Für Implementierungen gelten weiterhin folgende Regeln:

- Alle Felder einer Implementierung haben immer privaten Zugriff.
- Private Methoden müssen mit dem Zugriffsmodifizierer private gekennzeichnet werden. Methoden ohne Zugriffsmodifizierer werden als öffentlich angenommen und müssen damit in der öffentlichen Schnittstelle des implementierten Typs vorhanden sein. Zur öffentlichen Schnittstelle eines Typs zählen natürlich auch geerbte Methoden.
- Implementierungen sind innerhalb ihres eigenen Namensraums als Typen sichtbar. Damit ist folgender Quelltext gültig:

```
1  typeimpl ExampleImpl implements Example {
2    private static void staticTest() {
3      IO.println("static Hallo");
4    }
5    ExampleImpl() {
6    }
7    private void test() {
8      IO.println("Hallo");
9    }
10   void run() {
11     ExampleImpl impl = new ExampleImpl();
12     impl.test();
13     ExampleImpl.staticTest();
14   }
15 }
```

- Implementierungen eines package private-Typs müssen Mitglied desselben Pakets wie der implementierte Typ sein.

Mit folgenden Mitteln ist es möglich, Einfluss auf die Auflösung von Implementierungen zu nehmen:

.tjt-Datei: Für jede Implementierung, die der Compiler übersetzt, wird eine Datei erzeugt, welche dieser Implementierung einen Typ zuordnet. Die Dateien sollten nicht vom Nutzer verändert werden und sind die Grundlage der Implementierungsauflösung: Die Laufzeitumgebung scannt beim Starten die über -cp angegebenen Pfade (oder, falls keine angegeben wurden, das aktuelle Arbeitsverzeichnis) und sucht nach .tjt-Dateien. In einer .tjt-Datei darf nur eine Zuordnung der Form

<Type> = <Impl>

stehen. Kommentare sind erlaubt und werden durch # eingeleitet. Dabei ist <Type> der vollqualifizierte Name eines Typs und <Impl> der einfache Name einer Implementierung. Der vollqualifizierte Name der Implementierung wird aus dem Pfad zur .tjt-Datei gewonnen.

Konfigurationsdatei: Mit der Kommandozeilenoption -cfg kann eine Datei mit Typzuordnungen angegeben werden[11]. Sie hat folgende Syntax:

<Entry> { <newline> <Entry> }
<Entry> ::= <Typname> = <ImplName>

[11] Wird -cfg weggelassen, so wird versucht ./teajay.conf zu lesen.

Dabei sind <Typname> und <ImplName> vollqualifizierte Namen. Die Zuordnungen in dieser Datei überschreiben die vom Laufzeitsystem selbst gefundenen Implementierungen.

Die Kommandozeilenoption -Ximpl : Diese hat Vorrang vor den beiden oben genannten Methoden, darf beliebig oft Vorkommen und hat folgende Syntax:

```
> tj -Ximpl <Type>=<TypeImpl> ...
```

Damit lassen sich verschiedene Implementierungen bequem testen.

TypeManager.setTypeImpl(Class, String): Diese Methode erlaubt es die Zuordnung von Implementierungen zu Typen während der Laufzeit zu verändern. Alle bereits existierenden Instanzen behalten dabei ihre Implementierung, aber statische Methodenaufrufe werden sofort umgeleitet. Das Benutzen dieser Methode kann gefährlich sein und zu undefiniertem Verhalten führen: Der statische Kontext der vorherigen Implementierung wird, aus Sicht des Klienten, durch einen neuen ersetzt. Daher kann es passieren, dass eine Methode, ohne Verschulden des Entwicklers, seinen Vertrag nicht mehr einhält (wenn die Einhaltung etwa von statischen Feldern abhängt). Dies wird an folgendem Beispiel deutlich:

```
typedef SomeType {
  SomeType();
  /* gibt die Anzahl der Instanziierungsversuche zurück */
  static Number countInstances();
  void run();
}
typeimpl SomeTypeImpl implements SomeType {
  static Number instances = 0;
  SomeTypeImpl() {
    instances += 1;
  }
  static Number countInstances() {
    return instances;
  }
  void run() {
    //ruft countInstances() über die öffentliche Schnittstelle auf
    /* Aufruf 1 */ IO.println(SomeType.countInstances());

    //umgeht die Laufzeitumgebung und ruft countInstances() direkt auf
    /* Aufruf 2 */ IO.println(SomeTypeImpl.countInstances());
  }
}
typeimpl SomeOtherImpl implements SomeType {
  static Number instances = 0;
  SomeOtherImpl() {
    instances += 1;
  }
  static Number countInstances() {
    return instances;
  }
  void run() {
    IO.println("When you're not looking at it, this fortune is written in FORTRAN.");
  }
}
```

Sollte die Implementierung von SomeType zur Laufzeit ersetzt werden (z.B. von SomeTypeImpl zu SomeOtherImpl), so funktioniert die Methode SomeType.countInstances() nicht mehr korrekt. Es werden dann auch bei **Aufruf 1** und **Aufruf 2** verschiedene Methoden aufgerufen.

TypeManager.setTypeImpl(Class, String) existiert nur zu Versuchszwecken und muss explizit, über die Kommandozeilenoption -enable-rir[12] der Laufzeitumgebung, aktiviert werden. Wenn -enable-rir nicht angegeben wurde, so wird

[12] rir - runtime implementation replacement

TypeManager.setTypeImpl(Class, String) stets eine Ausnahme werfen (TeaJayRuntimeError).

ImplementationFinder: TeaJays Laufzeitumgebung bietet eine Schnittstelle an, mit deren Hilfe über benutzerdefinierten Code nach Implementierungen gesucht werden kann:

```
1  public interface ImplementationFinder {
2      public <T> Class<? extends T> findImplementation(Class<T> type);
3  }
4  //Benutzung:
5  TypeManager.instance().setImplementationFinder(new ImplementationFinder() {...});
```

findImplementation(Class) wird immer dann aufgerufen, wenn die automatische Suche nach dem Typ type fehlgeschlagen ist. Das Standardverhalten, falls für einen Typ keine Implementierung gefunden wird, ist es, auf der Konsole nach einer Implementierung zu fragen. Da TeaJay momentan keine generischen Instanzmethoden unterstützt, kann man dieses interface nur in Java implementieren.

3.4.4. Interfaces

MB

Die Interfaces in TeaJay beschreiben nicht direkt instanzierbare Typen, also Typen, für die die TeaJay-Laufzeitumgebung keine Implementierung sucht. Aus diesem Grund spezifizieren Interfaces keine Konstruktoren. Die Interfaces in TeaJay sind vorteilhaft, da es Typen gibt, die über die gleichen Fähigkeiten verfügen, sie aber nicht über den gleichen Basistyp modelliert werden können. Zum Beispiel gilt für diese Typen:

```
1
2  public typedef Set<T>{
3    Set();
4    /* Fügt ein element der Menge hinzu */
5    void add(T element);
6    /*
7     * Sucht Element der Menge mit der übergebenen Eigenschaft
8     * und gibt diese zurück.
9     */
10    T getElement(ElementProperty f);
11 }
12 public typedef OrderedSet<T> extends Set<T> interfaces ↩
     ↪TJIterable<T>{
13   OrderedSet();
14 }
15 public typedef MyList<T> extends List<T> interfaces TJIterable<↩
     ↪T>{
16 }
```

Das die Typen MyList und OrderedSet beide die Möglichkeit anbieten sollen, dass durch ihre verwalteten Elemente iteriert werden kann. Das bedeutet, beide Typen können einen Iterator erzeugen, aber einen gemeinsamen Basistyp, der die Fähigkeit

der Erzeugung eines Iterators externalisiert, wäre hier nicht möglich, da OrderdSet von Set erbt und dieser eben keine Fähigkeit zur Iteration besitzen soll.

Beim Erben von Interfaces wird keine Implementierung vererbt, dies führt dazu, dass es möglich ist, von mehreren Interfaces mit sich überschneidenen Methodensignaturen zu erben. Die Implementierung muss alle Methoden der angegebenen Interfaces implementieren. Der Compiler prüft, ob die Typimplementierung die angegebenen Interfaces der Typdefinition implementiert.

Ein Interface wird durch diese Regel beschrieben:

[“public“] “interface“ <identifier> [<Generic>] [“extends“ <Type>] “ { <MethodDecl> } “ “

Die Methodendefinitionen werden so definiert wie bei den Typdefinitionen, allerdings sind statische Methoden und Konstruktoren nicht erlaubt. Beides führt zu Kompilierfehlern.

Die Interfaces dürfen weiterhin von anderen Interfaces erben und können zudem generisch sein. Im Folgenden findet sich ein Beispiel für die Definition eines Interfaces:

```
1 public interface TJIterable<T>{
2    TJIterator<T> iterator();
3 }
```

Interfaces können nicht direkt instanziert werden, das heißt, es ist nicht möglich, in TeaJay Folgendes zu schreiben:

```
1   TJIterable<Number> iter = new TJIterable<Number>();
```

Allerdings lässt sich ein Typ, der das Interface bei seiner Typdefinition angibt, einem Interface zuweisen. Zum Beispiel:

```
1   TJIterable<Number> iter = new MyList<Number>();
2   //aber auch die Zuweisung
3   TJIterable<Number> orderedSet = new OrderedSet<Number>();
```

3.4.5. Closures

MK

Closures sind *Datentyp[en] für Routinen* [BSW10, S. 7] und kapseln eine anonyme Methode zusammen mit einem Erstellungskontext. Dabei wird in TeaJay der statische Typ eines Closures vollständig durch die Signatur der anonymen Methode beschrieben.

Dem parametrisierbaren TeaJay-Typ Closure ist es erlaubt beliebig viele Typargumente zu akzeptieren. Dabei werden die Parametertypen durch eine spezielle Syntax vom Rückgabetyp abgegrenzt. Um eine Variable als Closure zu deklarieren kann der Typ Closure wie folgt verwendet werden:

"Closure" [“<“<Type>“>“] "(" [<Type> {, <Type> }] ")"

Die Regel “<“<Type>“>“ steht dabei für den Rückgabetyp.

Bei der Erzeugung eines Closures gibt es in TeaJay eine Besonderheit: Der Erstellungskontext muss explizit angegeben werden. Das heißt ein TeaJay-Closure fängt

seine Umgebung nicht automatisch ein. Zudem ist es möglich, eigene private Felder im Erstellungskontext zu verwalten. Des Weiteren ist die anonyme Methode nicht Teil des privaten Namensraums einer umgebenden Implementierung und so ist es einem Closure nicht erlaubt, private Methoden jener aufzurufen. Soll aus einem Closure heraus eine Instanzmethode der öffentlichen Schnittstelle eines umgebenden Typs aufgerufen werden, so muss eine Referenz auf die Instanz des Typs explizit in den Erstellungskontext aufgenommen werden. Die o.g. Entwurfsentscheidungen für Closures ergaben sich aus TeaJays Fokussierung auf Schnittstellen: Closures grenzen sich durch ihre private Umgebung von der sie umgebenden Implementierung ab und können ebenfalls nur über öffentliche Schnittstellen mit anderen Typen kommunizieren.

Ein Closure kann in TeaJay nach folgender Regel erzeugt werden:

"closure" [<Env>] [“<“<Type>“>“] <Parameters> <Block>
<Env> ::= “[“ [<VarDeclarationWithInit> {";" <VarDeclarationWithInit>} ";"] “]“
<VarDeclarationWithInit> ::= <Type> <Identifier> "=" <Expression>
<Parameters> ::= "(" [<Param> {, <Param> }] ")"
<Param> ::= <Type> <Identifier>

Der Erstellungskontext wird durch (<Env>) bekanntgegeben und muss dort auch initialisiert werden. Die Ausdrücke innerhalb von <Env> werden im Umgebungskontext ausgewertet. Das heißt dort können Felder der Implementierung und private Methoden sowie lokale Variablen der umgebenden Methode genutzt werden. Der Erstellungskontext ist anschließend innerhalb des Closures sichtbar und bleibt über die gesamte Lebensdauer erhalten. Innerhalb des Closures kann man auf die Felder des Erstellungskontext zugreifen (lesend sowie schreibend):

```
1  Number num = 5;
2  Closure() cl1 = closure[Number num = num;]() {
3    IO.println(num); num += 1;
4  };
```

Im Beispiel wurde ein Closure ohne Rückgabewert und Parameter erzeugt und der Variablen cl1 zugewiesen. Der optionale Rückgabetyp eines Closures wird durch “<“<Type>“>“ bekanntgegeben und die Parameterliste (<Parameters>) gibt die Typen und Bezeichner der Argumente des Closures an:

```
1  Closure<Boolean>(String) cl2 = closure<Boolean>(String a) {
2    return a == "a";
3  };
```

Obiges Closure hat den Rückgabetyp Boolean und akzeptiert einen String als Argument.

Ein Closure kann über den Namen einer Variablen vom Typ Closure, oder direkt bei der Erzeugung, aufgerufen werden:

```
1  if (cl2("a")) { //Closure-Aufruf über Variable
2    IO.println(true);
3  }
4  //direkter Closure-Aufruf
5  String str = closure<String>() {
6    return "I haven't lost my mind -- it's backed up on tape somewhere.";
7  }();
```

An diesem Beispiel lässt sich auch erkennen, dass eine Variable vom Typ Closure gleichnamige Methoden verdeckt, denn syntaktisch lässt sich ein Closure-Aufruf nicht von einem Methodenaufruf unterscheiden. Dadurch vermischt sich der Namensraum

der Variablen teilweise mit dem Namensraum der Methoden. Diese Tatsache wurde hingenommen, da diese Form des Closure-Aufrufs von den TeaJay-Entwicklern als sehr intuitiv empfunden wurde.

Closures sollen es vereinfachen funktionale Programme zu formulieren: Durch Closures lassen sich viele Idiome der funktionalen Programmierung leichter formulieren und auch Methodenzeiger lassen sich mit ihrer Hilfe umsetzen. Mehr zu diesem Thema findet sich in Abschnitt 4.1.3. Closures dienen aber noch einem weiteren Zweck in TeaJay: Sie sollen verhindern, dass es notwendig wird überflüssig viele, kleine Typdefinitionen und Implementierungen zu erstellen. Möchte man z.B. Callbacks in TeaJay realisieren, so würde man ohne Closures z.B. wie folgt vorgehen müssen:

```
1  public typedef SomeType {
2    ...
3    void addEventListener(EventListener lstner);
4    ...
5  }//Dateiende
6  public interface EventListener {
7    void event();
8  }//Dateiende
9  public typedef MyEventListener interfaces EventListener {
10   MyEventListener();
11 }//Dateiende
12 typeimpl MyEventListenerImpl implements MyEventListener {
13   Number n = 1;
14   MyEventListenerImpl() {}
15   void event() {
16     IO.println("event " + n); n += 1;
17   }
18 }//Dateiende
19 //Code
20 SomeType a = ...;
21 a.addEventListener(new MyEventListener());
```

Closures können hier viele Zeilen Quellcode sparen und sollen dafür sorgen, dass Entwickler nicht von TeaJays restriktivem Typsystem frustriert werden:

```
1  public typedef SomeType {
2    ...
3    void addEventListener(Closure() lstner);
4    ...
5  }
6  //Code
7  SomeType a = ...;
8  a.addEventListener(closure[Number n = 1;]() {
9    IO.println("event " + n); n += 1;
10 });
```

Closures sind in TeaJay Referenztypen wie jeder andere Typ. Das heißt insbesondere, dass sie die Schnittstelle von TJObject implementieren:

```
1  String str = cl1.toString();
2  IO.println(str); //Ausgabe: Closure()
3  IO.println(cl1 == cl2); //Ausgabe: false
```

Die Methode operator_eq(Object) wurde dabei von TJObject geerbt und prüft auf referenzielle Gleichheit. Zudem kann sich ein Closure mittels this(...) rekursiv aufrufen:

```
1  Closure<Number>(Number) fibo = closure<Number>(Number n) {
2    if (n == 0 || n == 1) {
3      return n;
4    } else {
5      return this(n - 1) + this(n - 2);
6    }
7  };
8  IO.println(fibo(7));
```

Es gibt eine Ausnahme von der Regel, dass Closures nicht automatisch ihre Umgebung einfangen: Closures teilen sich automatisch alle vorhandenen Typvariablen mit ihrer Umgebung. Folgendes Beispiel macht deutlich, dass Typvariablen an jeder Stelle im Closure genutzt werden könnnen:

```
1  private static <T> void genericMethod(T a) {
2    Closure<T>(T) cl3 = closure[T c = a;]<T>(T a) {
3      T b = a; return b;
4    };
5  }
```

Ein Closure kann aber selber keine Typvariablen definieren.

3.4.6. Operatoren

Um die Anzahl der Operatoren klein zu halten, bietet TeaJay bewusst nur eine kleine Untermenge der aus Java bekannten Operatoren. Die weggelassenen Operatoren haben nach Meinung der Entwickler eine zu spezielle Bedeutung, um als überladbare Operatoren zur Verfügung zu stehen (wie z.B. die Bitshift-Operatoren <<, >>). Aus dem gleichen Grund sind alle logischen Operatoren in TeaJay nicht überladbar.

3.4.6.1. Nicht überladbare Operatoren

&& : Der binäre Operator && hat den Rückgabetyp Boolean und steht für das logische Und. Er arbeitet nur auf Boolean-Typen und hat die sogenannte Kurzschlusssemantik. Das heißt seine Auswertung wird gestoppt, sobald das Ergebnis des Ausdrucks feststeht. Dies wird an einem Beispiel deutlich:

```
1  Number a = ...;
2  if (a != null && a.isInteger()) { //kein Fehler
3    //tue etwas
4  }
```

Für den Fall, dass a null ist, wird a.isInteger() nicht ausgeführt. Die Auswertung geschieht von links nach rechts und daher würde folgender Code im Falle a == null fehlerhaft sein:

```
1  Number a = null;
2  if (a.isInteger() && a != null) { //wirft NullPointerException
3    //tue etwas
4  }
```

|| : Der binäre Operator || hat den Rückgabetyp Boolean und steht für das logische Oder. Er arbeitet ebenfalls nur auf Boolean-Typen und hat die Kurzschlusssemantik:

```
1  Number a = null;
2  if (true || a.isInteger()) { //kein Fehler
3    //Tue etwas
4  }
```

a.isInteger() wird in dem Beispiel niemals ausgeführt, da nach dem Auswerten von true der Wert des Ausdrucks bereits feststeht.

! : Der unäre Operator ! hat als Rückgabetyp Boolean und akzeptiert als Argument nur einen Boolean. Er wird wie folgt benutzt

!<Expression>

und negiert das Ergebnis von <Expression>.

= : Der Zuweisungsoperator ist ein binärer Operator. Sein Rückgabetyp ist der Typ seines rechten Operanden (R-Value). Er hat die Nebenwirkung, dass eine Referenz auf sein R-Value dem linken Operanden (L-Value) zugewiesen wird. Da in TeaJay tatsächlich alle Typen Referenztypen sind, ist der eigentliche Wert des R-Value bereits eine Referenz und diese wird nur ins L-Value kopiert. Hier die Syntax für =:

<L-Value> { *[* <Expression> *]* } = <R-Value> *;*

L-Value kann eine lokale Variable, ein Feld, eine lokale Variablendeklaration oder eine Felddeklaration sein. R-Value darf ein beliebiger Ausdruck sein. Der Typ von R-Value muss dabei zuweisungskompatibel (siehe Abschnitt 3.4.3.6) zum Typ des L-Value sein.

Cast-Operator : Der Cast-Operator hat folgende Syntax:

(<Type> *)* <Expression>

Als Rückgabewert hat der Cast-Operator eine Referenz auf den Wert von <Expression>. Sein Rückgabetyp ist <Type> und er hat die Nebenwirkung eine Ausnahme zu werfen, sollte der Typ T_e von <Expression> nicht zu <Type> konvertierbar sein. Dabei sollte <Type> ein Ober- oder Untertyp von T_e bzw. ein interface sein und ein Kompilierzeitfehler erzeugt werden, wenn dies nicht zutrifft. Der aktuelle TeaJay-Compiler setzt dies jedoch noch nicht durch. Es ist nicht erlaubt den Namen einer Implementierung für <Type> zu nutzen. Der Cast-Operator sollte in TeaJay selten gebraucht werden und überall, wo es möglich ist, durch typeswitch ersetzt werden.

3.4.6.2. Überladbare Operatoren

Operatoren sind Teil der öffentlichen Schnittstelle und müssen somit in der Typdefinition angegeben werden. TeaJay bietet folgende überladbaren Operatoren:

+, *, -, / : Arithmetische Operatoren werden mit folgenden Methodendeklarationen überladen:

```
1    static X operator_add(Y y, Z z);  // +
2    static X operator_mult(Y y, Z z); // *
3    static X operator_sub(Y y, Z z);  // -
4    static X operator_div(Y y, Z z);  // /
```

Dabei muss Y oder Z mit dem die Methode deklarierenden Typ übereinstimmen. X darf ein beliebiger Typ außer void sein. Die Suchreihenfolge der Methoden ist von links nach rechts, jedoch wird immer der beste Treffer aus beiden Typen A und B ausgewählt. Dies wird an einem Beispiel deutlich:

```
1    typedef A {
2      static A operator_add(A x, A y);
3    }
4    typedef B extends A {
5      static B operator_add(B x, A y);
6      static B operator_add(A x, B y);
7      static B operator_add(B x, B y);
8    }
9    //Code:
```

```
10      A a = ...;
11      B b = ...;
12      B c = a + b; //Ruft B.operator_add(A, B) auf
13      //<=>
14      B c = B.operator_add(a, b);
```

Zuerst wird in A eine aufrufbare Methode operator_add(A, B) gesucht und auch gefunden: operator_add(A, A). Anschließend wird trotzdem in B nach einem besseren Treffer gesucht und operator_add(A, B) gefunden. Für den Fall, dass in A und B zwei gleichwertige Methoden gefunden werden, wird die zuerst gefundene ausgewählt. Für die restlichen Arithmetik bzw. Vergleichsoperatoren gelten, mit Ausnahme des Methodennamens, die gleichen Regeln.

- (unär) : Wird der Operator - unär genutzt, so wird er durch folgende Methodendeklarationen überladen:

```
1       static X operator_sub(X z);
```

Dabei muss X mit dem die Methode deklarierenden Typ übereinstimmen. Ein Beispiel verdeutlicht dies:

```
1       Number a = -5;
2       // <=>
3       Number a = Number.operator_sub(5);
```

+=, *=, -=, /= : Diese Operatoren werden gleichzeitig mit ihren Hauptoperatoren +, *, - bzw. / automatisch überladen. Sie haben folgende Semantik (dabei steht ∘ für einen der vier Hauptoperatoren):

$$\mathsf{a} \circ= \ldots; \iff \mathsf{a} = \mathsf{a} \circ (\ldots);$$

<, >, <=, >= : Vergleichsoperatoren werden mit folgenden Methodendeklarationen überladen:

```
1       static Boolean operator_less(Y y, Z z);      // <
2       static Boolean operator_greater(Y y, Z z);   // >
3       static Boolean operator_leq(Y y, Z z);       // <=
4       static Boolean operator_geq(Y y, Z z);       // >=
```

Hier gilt auch wieder, dass Y oder Z mit dem die Methode deklarierenden Typ übereinstimmen soll. Es gilt die gleiche Suchreihenfolge wie für die arithmetischen Operatoren.

== und != : Der Vergleichsoperator unterscheidet sich stark von den vorangegangenen Operatoren: Er wird von einer Instanzmethode überladen und daher vererbt. Es gibt auch nur eine gültige Signatur:

```
1       Boolean operator_eq(Object other);
```

Der Vergleichsoperator ist kompatibel mit Java-Typen und ruft dann Object.equals(Object) auf. Eine weitere Besonderheit ist, dass der Vergleichsoperator für TeaJay-Typen in beiden Richtungen prüft: Sollte a.operator_eq(b) false liefern, wird auch noch b.operator_eq(a) überprüft. Auch wird operator_eq

niemals für den Vergleich mit null aufgerufen. Das heißt kein Typ außer null selbst kann behaupten äquivalent zu null zu sein. Das ermöglicht es Folgendes zu schreiben:

```
String a = ...;
if (a == null) {
  throw new NullPointerException("a may not be null");
}
...
```

Der Operator != ist automatisch definiert als die Negation von ==:

$$\mathsf{a} \mathrel{!}= \mathsf{b} \iff !(\mathsf{a} == \mathsf{b})$$

Um auf referenzielle Gleichheit zu prüfen, gibt es in TeaJay keinen Operator, stattdessen dient die Methode teajay.util.TJObjects.referenceEquals(Object, Object) diesem Zweck.

[ ] : Der Zugriffsoperator vereint eigentlich zwei verschiedene Operatoren in einem Symbol. Je nach Kontext hat er eine andere Bedeutung:

[ ] als R-Value: Der R-Value Operator [ ] wird mit folgender Methodendeklaration überschrieben:

```
X operator_get(Y y);
```

Dabei dürfen X und Y beliebige Typen sein. Seine Semantik kann wie folgt beschrieben werden:

$$a[b_0]\ldots[b_n]$$
$$\iff$$
$$a.\mathsf{operator\_get}(b_0).(\ldots).\mathsf{operator\_get}(b_{n-1}).\mathsf{operator\_get}(b_n)$$

a und b_i stehen für beliebige Ausdrücke.

[ ] als L-Value: Der L-Value Operator [ ] wird mit folgender Methodendeklaration überschrieben:

```
void operator_set(Y y, X x);
```

Hier dürfen ebenfalls X und Y beliebige Typen sein. Seine Semantik kann wie folgt beschrieben werden:

$$a[b_0]...[b_n] = c$$
$$\iff$$
$$a.\mathsf{operator\_get}(b_0).(...).\mathsf{operator\_get}(b_{n-1}).\mathsf{operator\_set}(b_n\ ,\ c)$$

Dabei stehen a, b_i und c für beliebige Ausdrücke.

3.4.6.3. Operatorvorrang

Folgende Tabelle beschreibt die Vorrangregeln für Operatoren in TeaJay. Je niedriger der Rang desto höher der Vorrang:

Rang	Operator	Beschreibung	Arität / Notation
1	! - (<Type>)	logische Negation arithmetische Negation Typisierung (typecast)	unär / Präfix
2	* /	Multiplikation Division	binär / Infix
3	+ -	Addition Subtraktion	binär / Infix
4	< > <= >=	kleiner als größer als kleiner gleich größer gleich	binär / Infix
5	== !=	Gleichheit Ungleichheit	binär / Infix
6	&& ||	logisches Und logisches Oder	binär / Infix
7	= += *= -= /=	Zuweisung Additionszuweisung Multiplikationszuweisung Subtraktionszuweisung Divisionszuweisung	binär / Infix

Tabelle 3.1.: Operatorvorrang in TeaJay

3.4.6.4. Abwägungen zu Operatoren und Operatorüberladung

Eine der Anforderungen an TeaJay ist, dass benutzerdefinierte Typen sich weitestgehend natürlich in der Sprache nutzen lassen. Eingebaute Typen sollen keine all zu große Sonderstellung haben (z.B. dass die arithmetischen Operatoren nur für Number verfügbar sind). Auf arithmetische und vergleichende Operatoren ganz zu verzichten kam für die Entwickler nicht in Frage, da dadurch die Lesbarkeit von TeaJay-Quelltext stark leiden würde:

```
1  //Ohne Operatoren:
2  Number sum = n.mult(n.add(1)).div(2);
3  //Mit Operatoren:
4  Number sum = n * (n + 1) / 2;
```

TeaJay unterstützt daher für ausgewählte Operatoren die Operatorüberladung. Die eingebauten Typen haben aber in TeaJay trotzdem eine gewisse Sonderstellung durch die nicht überladbaren Operatoren: Logische Operatoren arbeiten beispielsweise nur auf dem eingebauten Typ Boolean. Auch ist es den eingebauten Typen vorbehalten den Zuweisungsoperator zu überladen:

```
1  String str = "hallo";
2  Number n = 5;
3  Boolean bool = true;
```

Dieses Verhalten kann man für benutzerdefinierte Typen nicht nachbauen. Durch überladene Operatoren kann jedoch auch die Lesbarkeit von Quelltext leiden. Sei der Typ Complex wie folgt definiert:

```
typedef Complex {
  static Complex operator_add(Number a, Complex b);
  static Complex operator_add(Complex a, Number b);
  static Complex operator_add(Complex a, Complex b);
  ...
}
```

Es ist dann nicht auf den ersten Blick ersichtlich, welchen Typ folgender Ausdruck hat:

$$5 * b + 3 - a$$

Sollte a oder b vom Typ Complex sein, so ist der Typ des Ausdrucks Complex. Sollten a und b vom Typ Number sein, so hat der Ausdruck den Typ Number. Die gleiche Kritik würde jedoch auch auf den Ansatz, nur Methoden zu nutzen, zutreffen.

Arithmetik- und Vergleichsoperatoren generieren beim Leser auch häufig bestimmte Erwartungen: Zum Beispiel würde man vermutlich erwarten, dass + kommutativ ist oder, dass für < und >=

$$!(a < b) \Leftrightarrow a >= b$$

gilt. Eine weitere Erwartung könnte sein, dass diese Operatoren nebenwirkungsfrei sind. Ein Implementierer könnte aber all diese Erwartungen verletzen. Überlegt genutzt kann die Operatorüberladung jedoch förderlich für die Lesbarkeit sein und ihr Einsatz sollte auch einen Platz in der Lehre haben.

Beim Entwurf von TeaJay wurde auch der Zugriffsoperator ([ ]) kontrovers diskutiert. Fast jede verbreitete imperative Sprache besitzt diesen Operator. Seine Aufnahme in den Sprachumfang von TeaJay hat daher eher pragmatische Gründe: Es sollten Reibungsverluste beim Umstieg von TeaJay auf andere Sprachen vermieden werden. Er bringt jedoch auch überflüssige Syntax in die Sprache und macht sie damit evtl. schwerer zu erlernen.

Der Cast-Operator ist, durch typeswitch, auf den ersten Blick überflüssig. typeswitch führt jedoch nur sichere Typkonvertierungen durch und arbeitet nur eingeschränkt auf parametrisierten Typen. Mit dem Cast-Operator ist es möglich typunsichere Konvertierungen durchzuführen. Die Notwendigkeit typunsicherer Konvertierungen liegt am generischen Typsystem von TeaJay:

```
List<?> anyList = ...; //z.B. könnte hier eine serialisierte Liste eingelesen werden.
//Vorsicht: Der folgende Cast wird keine ClassCastException werfen.
List<String> stringList = (List<String>) anyList;
assert stringList.size() > 0;
String a = stringList.get(0); //Kann eine ClassCastException werfen, wenn anyList keine List<↩
    ↪String> war.
```

In dem Beispiel wird List<?>, mit unbekanntem Inhalt, zu einer List<String> umgewandelt. Es kann hier aber weder zur Kompilierzeit noch zur Laufzeit überprüft werden, ob anyList wirklich eine List<String> ist. Das liegt an der Umsetzung des generischen Typsystems: Wie bei Java werden in TeaJay die generischen Typinformationen nur zur Kompilierzeit überprüft und liegen zur Laufzeit nicht mehr vor (siehe Abschnitt 3.4.3.7). Oben gezeigtes Phänomen nennt die JLS *heap pollution* [GJS+12, Paragraph 4.8].

3.4.7. Blöcke und Statements

3.4.7.1. Block

MB

In strukturierten Programmiersprachen gibt es fast immer die Möglichkeit Quelltextabschnitte zu Blöcken zusammenzufassen. Die Art der Blockbegrenzung nimmt Einfluss auf die Wahrnehmung von Quelltext und kann den Programmierer auch zu einer gewissen Disziplin bei der Strukturierung zwingen. Für TeaJay wurden verschiedene Vorgehensweisen diskutiert:

- **Tabulatorblöcke**: Diese werden z.B. in Python verwendet. Der Vorteil dieser Art der Blockbegrenzung ist, dass der Entwickler gezwungen ist, sein Programm für die meisten Leser gut zu strukturieren. Der Nachteil ist, Tabulatoren sind Whitespace-Zeichen und können nicht von Leerzeichen unterschieden werden. Dadurch kann es zu technischen Problemen kommen, da Tabulatoren jeweils nach dem verwendeten Editor und seinen Einstellungen als ein Steuerzeichen \t oder als Leerzeichen mit z.B. vier oder acht Zeichen dargestellt werden können. Diese Problematik hat störenden Einfluss auf die Programmierung.

- **Geschweifte Klammern**: Werden vor allem von Programmiersprachen verwendet, die von der Programmiersprache C abstammen. Der Vorteil von geschweiften Klammern ist, dass sie dem Entwickler eine gewisse Freiheit bei der Strukturierung des Quelltextes einräumen. Allerdings strukturieren die meisten Entwickler ihren Quelltext mit geschweiften Klammern und Tabulatoren, um die Lesbarkeit zu erhöhen. Der Nachteil ist, dass bei nicht disziplinierter Handhabung keine Lesbarkeit mehr gegeben ist.

- **Begin und End**: Die aus Pascal bekannten Blockbegrenzer haben die gleichen Vorteile und Nachteile wie die geschweiften Klammern. Jedoch ist ein weiterer Kritikpunkt, dass viele Zeichen verwendet werden müssen, um etwas Simples auszudrücken.

Blöcke in TeaJay sind nach folgender Regel definiert:

<Block>::="{" { <Statements> } "}"

Folglich werden geschweifte Klammern genutzt, da TeaJay eine Sprache ist, die von Java erbt und dem Entwickler Freiheiten bei der Strukturierung von Quelltext gewähren möchte.

3.4.7.2. Expression Statements

Bei den Expression-Statements wurde darüber diskutiert, wie das Anweisungsende kenntlich gemacht werden sollte. Es wurde die Möglichkeit erwogen, ein Symbol zu verwenden oder dass der Zeilenumbruch das Ende der Anweisung[13] markiert. In TeaJay ist die Erwartung, dass Bezeichner und Ausdrücke sehr lang werden und deshalb

[13] Auch ein Zeilenumbruch wäre ein Symbol, nur nicht sichtbar

Ausdrücke umgebrochen werden müssen, damit das Lesen von Quelltext ohne horizontales Scrollen ermöglicht wird. Deshalb kann die Lesbarkeit von Quelltext durch die Verwendung von Zeilenumbrüchen verschlechtert werden. Dem gegenüber liegt die Gefahr bei der Verwendung eines Symbols, dass mehrere Expression-Statements in eine Zeile geschrieben werden. Dies ist aber nur bei undisziplinliertem Verhalten gegeben.

Für die Verwendung eines Symbols zum Markieren eines Anweisungsendes spricht, dass in natürlichen Sprachen Punkte, Semikolons, Ausrufezeichen u.s.w. Verwendung finden, um zu verdeutlichen, dass ein Satz bzw. eine Anweisung beendet ist. Dies sollte gerade Anfängern helfen. Die Expression-Statements wurden daher wie in Java definiert:

<StatementExpression>::=<Expression> ";"

3.4.7.3. Lokale Variablendeklarationen

Beim Entwurf der lokalen Variablendeklarationen wurden zwei Aspekte diskutiert. Die erste zu klärende Frage war, ob bei der Deklaration einer Variablen der Typ angegeben werden soll, während es beim zweiten Aspekt darum ging, welchen Wert eine Variable besitzt, die bei ihrer Deklaration nicht initialisiert wird.

Die nicht-Angabe des Typs ist für eine Lehrsprache ungeeignet, da die Lernenden ansonsten aus dem Kontext heraus den Typ einer Variablen ermitteln müssen. Zur Klärung der zweiten Frage sind verschiedene Alternativen denkbar. Diese sind:

1. Der Wert einer Variablen ist undeterminiert. Der Nachteil ist, dass Zugriffe auf die Variable zu schwer nachvollziehbaren Laufzeitfehlern führen können und es keine Möglichkeit gibt zu prüfen, ob die Variable uninitialisiert ist.

2. Der Standardkonstruktor des Typs der Variablen wird aufgerufen. Der Vorteil bei dieser Vorgehensweise ist, dass es so möglich ist, dass jede Variable in einem kontrollierbaren Zustand ist. Allerdings muss jeder Typ dann einen Standardkonstruktor besitzen. Dies ist aber nicht für alle Typen sinnvoll.

3. Die Variable wird mit null initialisiert. Der Vorteil ist, dass bei der Verwendung der Variablen der Laufzeitfehler leicht verständlich ist. Weiterhin ist die Prüfung auf null durch den Entwickler möglich.

4. Der Compiler überprüft vor der Verwendung, ob die Variable initialisiert ist. Der Vorteil ist, dass der Compiler anstelle des Laufzeitsystems dem Entwickler eine Rückmeldung gibt. Das heißt, nicht initialisierte Variablen werden früh gefunden. Der Nachteil ist, dass durch die Prüfung zur Laufzeit der Quelltext für den Compiler angepasst werden muss, so dass er auf nicht-Initialisierung prüfen kann. In Java lehnt der Compiler den folgenden Quelltext ab:

```
1 int a = 1, b;
2 if(a > 0) { b = 1; }
3 if(a <= 0){ b = 2; }
4 System.out.println(b);
```

Dagegen wird dieser Quelltext vom Compiler akzeptiert:

```
1 int a = 1, b;
2 if(a > 0){ b = 1; }
3 else     { b = 2; }
4 System.out.println(b);
```

Beide Quelltexte führen das Gleiche aus, dabei kann der Compiler im ersten Fall nicht bestimmen, ob das Programm eine der beiden if-Statements betritt. Hingegen ist der Compiler im zweiten Fall in der Lage zu bestimmen, dass b in jedem Fall initialisiert wird, da entweder der if-Block oder der else-Block betreten wird. Allerdings bedeutet dies, dass der Compiler Fehler meldet, obwohl kein Fehler aufgetreten ist.

Für TeaJay ist die dritte Alternative gewählt worden. Die lokale Variablendeklaration ist wie folgt definiert:

<LocalVariableDeclaration> “;“
<LocalVariableDeclaration>::= <Type> <VariableDeclarators>
<VariableDeclarators>::= <Identifier> [<VarInit>] { VarDeclarator }
<VarInit>::= “=“ <Expression>
<VarDeclarator>::= “,“ <Identifier> [<VarInit>]

Nachfolgend wird ein Beispiel von lokalen Variablendeklarationen dargestellt:

```
1 Number a = 1;
2 Number b;
3 Number c, d= 12, e;
4 IO.println(a); // Gibt 1 auf der Konsole aus
5 IO.println(b); // Gibt null auf der Konsole aus
```

Für die lokale Variablendeklarationen gelten folgende Regeln, die bei Nichteinhaltung einen Kompilierfehler ergeben:

- Die Selbstzuweisung ist nicht möglich.
- Der Typ des Ausdrucks muss mit dem Typ der Variablen zuweisungskompatibel sein.
- Die Deklaration von lokalen Variablen mit gleichem Namen innerhalb eines Blocks oder seiner inneren Blöcke ist nicht möglich.
- Zugriffe auf lokale Variablen sind nur innerhalb seines deklarierenden Blocks und in seinen verzweigenden Blöcken möglich.

Des Weiteren hat die lokale Variablendeklaration im Zusammenhang mit Felddeklarationen die Eigenschaft, dass sie Felddeklarationen verdeckt.

3.4.7.4. assert-Statement

MK

Das assert-Statement

"assert" <Expression> [":" <Expression>]

drückt bestimmte Annahmen bzw. Zusicherungen aus, die der Entwickler an dieser Stelle des Progammflusses macht. Es hat die Nebenwirkung diese auch zu überprüfen und den Programmfluss mit einer Ausnahme zu unterbrechen, falls der erste, nicht optionale, Ausdruck zu false auswertet. Der zweite, optionale, Ausdruck stellt eine Fehlerbeschreibung dar. Während der Ausdruck, welcher den Fehler beschreibt, jeden TeaJay-Typen außer void haben darf, ist es ein Fehler, wenn der erste Ausdruck nicht vom Typ Boolean ist. Bei der Benutzung von assert gilt es folgendes zu beachten:

- Die beiden Ausdrücke sollten keine Nebenwirkungen haben, die den Programmfluss in irgend einer Weise verändern können, damit ein Deaktivieren der Assertionprüfungen nicht zu schwer lokalisierbaren Fehlern führt.
- Der zweite Ausdruck wird unabhängig davon, ob die Zusicherung zutrifft, immer ausgewertet.
- Wenn der zweite Ausdruck mit java.lang.Throwable zuweisungskompatibel ist, wird der Rückgabewert als Grund für den java.lang.AssertionError eingetragen. (siehe java.lang.Throwable(Throwable cause))
- Sollte obiges nicht der Fall sein, wird die String-Repräsentation des Rückgabewertes als Fehlerbeschreibung des java.lang.AssertionError verwendet.
- Mit dem aktuellen Compiler ist das Deaktivieren der Überprüfung von assert-Statements noch nicht möglich.

Im folgenden ein kleines Beispiel:

```
1  Number b = 1;
2  Number a = b * b;
3  assert a >= 0 : "unmöglicher Fall"; // immer wahr
4  assert a < 0; // wirft immer AssertionError
```

Zusicherungen sind ein nützliches Werkzeug beim Entwickeln nach der Design-by-Contract-Methode[14]. Sie helfen auch dabei Fehler frühzeitig zu erkennen, die Gedanken des Programmierers besser nachvollziehen zu können und sind auch der erste Schritt Quelltext zu verifizieren. All diese Eigenschaften machen den Einsatz von Zusicherung in einer Lehrsprache sinnvoll.

3.4.7.5. Bedingte Anweisungen

MB

In Programmiersprachen muss es die Möglichkeit geben, dass Anweisungen nur unter bestimmten Bedingungen durchgeführt werden, ansonsten wäre die Sprache nicht mächtig genug, um mit ihr alle turingberechbaren Probleme zu lösen. In TeaJay existieren zwei Sprachelemente zur Durchführung von bedingten Anweisungen, diese sind:

if-Statement:

<If>::="if" "(" <Expression>")" <Block> [<Else>]
<Else>::="else" (<Block> | <If>)

[14]Weiterführende Informationen finden sich in [Mey92]

Bei der Benutzung von if ist folgender Aspekt zu beachten: Es ist ein Kompilierfehler, wenn der Ausdruck im Statement nicht vom Typ Boolean ist. Das folgende Beispiel zeigt die Benutzung des if-Statements:

```
1  Number b = 10;
2  if( b < 10 ){
3    IO.println("Wird nicht ausgeführt");
4  }else if( b > 10 ){
5    IO.println("Wird nicht ausgeführt");
6  }else{
7    IO.println("Wird ausgeführt");
8  }
```

typeswitch-Statement:

“typeswitch“ “(“ <Identifier> “)“ “{“ <Typeswitchcases> “}“
<Typeswitchcases> ::= { <Typeswitchcase> } “default“ <Block>
<Typeswitchcase> ::= “case “ <Type> <Block>

Das Statement wurde entworfen, um Zuweisungskompatibilität zwischen Laufzeittypen zu testen. Der Bezeichner von typeswitch muss eine lokale Variable sein[15]. Wenn ein passender Typ gefunden wurde, so wird nur der entsprechende Fallblock ausgeführt. Innerhalb des Ausführungsblocks ist die lokale Variable vom Typ, auf den geprüft wurde.

Die Motivation für typeswitch ist, dass es in objektorientierten Sprachen, die statisch typisiert sind, vorkommt, dass der Laufzeittyp ermittelt werden muss. Diese Prüfung wird in der Regel, z.B. in Java, durch folgenden Quelltext abgesichert und durchgeführt:

```
1 //func: Erzeugt eine Instanz von Student oder Mitarbeiter.
2 Person person = func();
3 if(person instanceof Student){
4   Student student = (Student)person;
5 }else if( person instanceof Mitarbeiter){
6   Mitarbeiter mitarbeiter = (Mitarbeiter) person;
7 }else{
8   System.err.println("Fehler");
9 }
```

Die Absicherung mittels if-else geschieht, damit die Umwandlung einer Person nicht zu einem Laufzeitfehler führt. Bei dieser Vorgehensweise ist anzumerken, dass die explizite Umwandlung innerhalb der Blöcke unnötig wäre, da ein Compiler erkennen könnte, dass die Zuweisung funktionieren müsste, weil der Compiler in diesem Fall den Laufzeittyp kennen könnte. Des Weiteren muss sich der Entwickler bei dieser Vorgehensweise selbst darum kümmern, dass er vom speziellsten zum allgemeinsten Typ hin prüft. Die Nichteinhaltung der Reihenfolge ist eine Fehlerquelle bei der Entwicklung.

[15]Zugriffe auf Felder werden im Ausblick dargestellt.

Durch die Verwendung von typeswitch ist die explizite Umwandlung in Fallblöcke nicht nötig und das Anordnen der Reihenfolge von speziell nach allgemein wird vom Compiler übernommen. Sind bei den Falltypen Interfaces vorhanden, so werden diese zuletzt geprüft. Bei den Interfaces wird, wie bei allen Typen auch, vom Speziellsten zum Allgemeinsten hin geprüft. Die Reihenfolge der Prüfung lässt sich durch den Ableitungsbaum leicht ermitteln. Für jeden Falltyp wird die Tiefe des Typs im Ableitungsbaum ermittelt. Dann lassen sich die Fälle anhand ihrer Tiefe absteigend sortieren. Des Weiteren ist der default-Fall nicht optional. Hierdurch muss der Entwickler bedenken, was passieren soll, wenn kein passender Typ gefunden wird. Dies wird oft bei der if-else-Vorgehensweise vergessen. Ein weiterer Vorteil ist, dass der Quelltext durch typeswitch strukturierter wirkt.

Eine weitere Eigenschaft von typeswitch ist, wie dieser mit parametrisierten Typen umgeht. Da zur Laufzeit keine generischen Typinformationen vorhanden sind, unterstützt die Anweisung keine Unterscheidung zwischen den gleichen generischen Typen mit unterschiedlicher Parametrisierung. Hierbei wird dieser Quelltext als Kompilierfehler behandelt:

```
1 Object a = new List<String>();
2 typeswitch(a){
3     case List<String>{ IO.println("String");}
4     default { }
5 }
```

Da es trotzdem möglich sein soll, mittels typeswitch auf einem generischen Typ zu prüfen, wird verlangt, dass der angegebene generische Falltyp mit ? parametrisiert wurde, wie hier gezeigt ist:

```
1 Object a = new List<String>("a");
2 typeswitch(a){
3     case List<?>{ Object obj = a.get(0);}
4     default { }
5 }
```

Durch diese Angabe ist es nur noch möglich, null in die Liste zu schreiben und java.lang.Objects aus der Liste zu Lesen.

Innerhalb eines Falls werden Zuweisungen an die geprüfte Variable der Anweisung unterbunden. Dies dient der Typsicherheit von parametrisierten Typen zur Laufzeit. Ansonsten wäre folgendes möglich:

```
1 List<String> a = new MyList<String>();
2 typeswitch(a){
3     case MyList<?>{ a = new MyList<Number>(5);
4     }
5     default { }
6 }
7 String str = a.get(0); // Laufzeitfehler
```

Im Folgenden wird ein Beispiel zur Nutzung von typeswitch gezeigt, welches aus [SDRS12] entnommen wurde. Beim präsentierten Beispiel geht es um die Evaluierung dieser Grammatik:

exp := value |<exp> + <exp> |<exp> * <exp> |<exp> - <exp> |<exp> \<exp>

Die Repräsentation der Grammatik sieht wie folgt aus:

```
1 typedef Exp{ Exp(); Number eval(); }
2 typedef Value extends Exp { Value(String n); Number getValue() }
3 typedef Mult extends Exp { Mult(Exp x, Exp y); Number getXExp(); Number getYExp(); }
4 ...
```

Eine Evalatorimplementierung kann wie folgt aussehen:

```
1
2  typeimpl EvaluatorImpl implements Evaluator {
3    Exp ast;
4    Evaluator(String c){
5    /*Baue Abstrakten Syntaxbaum aus dem String*/}
6
7    Number evaluate(){
8      return eval(ast);
9    }
10
11   private Number eval(Exp e){
12     typeswitch(e){
13       case Value { return e.getValue(); }
14       case Plus  { return eval(e.getXExp())+eval(e.getYExp() ); }
15       case Sub   { return eval(e.getXExp())-eval(e.getYExp() ); }
16       case Mult  { return eval(e.getXExp())*eval(e.getYExp() ); }
17       case Divide{ return eval(e.getXExp())/eval(e.getYExp() ); }
18       default {
19         IO.println("Fehler");
20       }
21     }
22   }
23 }
```

Quellcode 3.9: EvaluatorImpl.tj

Indizien für einen schlechten Entwurf können sein, dass Instanzen umgewandelt werden und die Ausführung explizit davon abhängt, von welchem Typ eine Instanz ist. Jedoch sind Typumwandlungen gerade in polymorphen Sprachen nicht einfach zu verhindern, denn dies erfordert beim Entwurf der Software ein hohes Maß an diszipliniertem Vorgehen. Allerdings bieten viele Programmiersprachen Möglichkeiten an, um eine explizite Konvertierung von Typen zu ermöglichen und deshalb kann es gut sein, dass Anfänger lernen, mit den Problematiken der Konvertierung von Typen umzugehen.

Ein Statement, das häufig in imperativen Programmiersprachen vorkommt, ist das switch-case-Statement. Es dient der Fallunterscheidung. Eine mögliche Grammatikregel für TeaJay könnte so aussehen:

“switch“ “(“ <Expression> “)“ “{“ <Cases> “}“
<Cases>::= { <Case> } [“default:“ <Block>]
<Case>::= “case“ <Expression> “:“ <CaseBlock>
<CaseBlock>::= “{“ { Statements } [<Break>] “}“

Das mögliche Verhalten des **switch-case**-Statements ist hier wie folgt beschrieben: der switch-Ausdruck wird ausgewertet und anschließend werden die case-Ausdrücke nacheinander geprüft. Bei Übereinstimmung wird der Fallblock ausgeführt, andernfalls wird der nächste case-Ausdruck überprüft. Sollte kein Fall zutreffen, so wird der default-Block ausgeführt.

Für das switch-case-Statement muss zuerst diskutiert werden, wie sich das Statement verhält, wenn ein Fallblock ausgewählt wird, der nicht mit einem break-Statement abschließt. Nach der Semantik von Java fällt die Ausführung bei fehlender break-Anweisung in den nächsten Fallblock durch. Aus der Perspektive eines Anfängers ist dies wahrscheinlich nicht das erwartete Verhalten, denn vermutlich nähme er an, dass genau ein Fall ausgeführt wird und dann die Bearbeitung des switch-case-Statements abgeschlossen ist.

Das zweite Problem ist die Frage, von welchem Typ und welcher Art die Fallausdrücke sein dürfen. In Java müssen die Fallausdrücke vom Typ int, dem Typ String oder ein Aufzählungstyp sein. Hinzu kommt noch, dass die Fallausdrücke konstant sein müssen. Des Weiteren können erfahrene Entwickler die Erwartung haben, dass mit Nutzung der switch-case-Anweisung auch ein gewisser Performanzgewinn erzielt wird. Diese Erwartung würde aus technischen Gründen in TeaJay nur mit dem Typ String und den Aufzählungstypen bestätigt. Denn beim Typ Number müsste die zu prüfende Variable eine ganze Zahl sein, die auch im Gültigkeitsbereich von int aus Java liegt. Hierbei kann die Typsicherheit aber nur zur Laufzeit zugesichert werden und erst dann könnte ein Entwickler die Meldung erhalten, dass ein Typfehler geschehen ist. Dies ist nur ein technischer Aspekt. Allerdings sprechen keine Gründe dagegen die Erwartungshaltung von erfahrenen Entwicklern in diesem Fall zu durchbrechen und alle Zahlkonstanten aus TeaJay zu erlauben.

Allerdings ist eine Forderung an TeaJay, dass sich die benutzerdefinierten Typen und die eingebauten Typen in ihren Fähigkeiten nicht stark unterscheiden, denn es gäbe keinen technischen Grund mehr dafür, die switch-case-Anweisung auf eine kleine Zahl von Typen zu limitieren. Jedoch würde dies dazu führen, dass die Fallausdrücke beliebig sein dürfen und dies möglicherweise zu schwer nachvollziehbarem Verhalten der Anweisung führen kann. Denn die Anweisung muss mit Fallausdrücken zurecht kommen, die identische Werte erzeugen. Des Weiteren muss auch festgelegt werden, zu welchem Zeitpunkt die Fallausdrücke zur Laufzeit ausgewertet werden sollen. Zwar kann durch die switch-case-Anweisung der Quelltext strukturierter wirken, sie lässt sich aber auch gut durch die if-Anweisung vermeiden. Durch diese Überlegungen existiert die switch-case-Anweisung in TeaJay nicht.

3.4.7.6. Schleifen-Statements

MK

Schleifen sind eine spezielle Form bedingter Anweisungen und heutzutage aus der imperativen Programmierung nicht mehr weg zu denken. Durch das Aufkommen der strukturierten Programmierung [Bö95] haben Schleifenanweisungen in beinahe jede imperative Sprache Einzug erhalten. In TeaJay gibt es drei Formen von Schleifen:

Die while-Schleife:

"while" "(" <Expression> ")" <Block>

Die while-Schleife wiederholt die Anweisungen in <Block> solange <Expression> zu true auswertet. Es ist ein Kompilierzeitfehler, wenn <Expression> nicht vom Typ Boolean ist.

Die do-while-Schleife:

"do" <Block> "while" "(" <Expression> ")" ";"

Die do-while-Schleife ist eine sogenannte fußgesteuerte Schleife: Das heißt sie führt alle Anweisungen, die in <Block> vorkommen, mindestens einmal aus und erst dann wird die Schleifenbedingung (<Expression>) überprüft. Auch hier muss <Expression> den Typ Boolean haben.

Obwohl eine Schleifenart für eine Programmiersprache ausreichen würde, wurde die do-while-Schleife in TeaJays Sprachumfang aufgenommen. Für die Aufnahme sprechen, dass do ... while (<Bedingung>) dem natürlichen Sprachgebrauch ähnelt und sich mit dem geschickten Einsatz von do-while Codedopplung vermeiden lässt.

Die for-Schleife:

"for" "(" <Type> <Identifier> ":" <Expression> ")" <Block>

TeaJays for-Schleife ist in Wirklichkeit eine for-each-Schleife(vgl. [for]). Es sei T der Typ des Bezeichners (also <Type>). Der Typ von <Expression> muss dann mit den Typen teajay.lang.TJIterable<? extends T> oder java.lang.Iterable<? extends T> zuweisungskompatibel sein. Eine Ausnahme bildet der Typ String, welcher zwar iterierbar ist, aber weder teajay.lang.TJIterable noch java.lang.Iterable erfüllt. Dies ist ein Artefakt technischer Natur und wird in Kapitel 5 weitergehend behandelt. Die Semantik von TeaJays for-Schleife kann man am besten mit TeaJay selber ausdrücken:

```
1  List<Number> lst = ...;
2  for (Number n : lst) {
3    IO.println(n);
4  }
5  /* Folgende while-Schleife ist äquivalent zu obiger for-Schleife. */
6  {
7    TJIterator<Number> iter = lst.iterator();
8    while (iter.hasNext()) {
9      Number n = iter.next();
10     {
11       IO.println(n);
12     }
13   }
14 }
```

Die Entwickler von TeaJay haben sich entschieden die for-Schleife in TeaJays Sprachumfang aufzunehmen, da das Iterieren über eine Ansammlung von Elementen ein häufiger Anwendungsfall ist. Die Bedeutung obiger Schleife lässt sich in Worten etwa so ausdrücken: Für alle Elemente in lst führe die Methode IO.println(n) aus, wobei n jeweils für das sich in Bearbeitung befindende Element steht.

TeaJay verzichtet bewusst auf eine for-Schleife der Form

```
1  for (Initialisierung; Bedingung; Fortsetzung) ...
```

wie sie z.B. in C, C++ und auch Java existiert. Die Entwickler waren in diesem Fall der Meinung, dass diese Schleifenform unnötige Komplexität in die Sprache bringt und vermutlich nur von erfahrenen Programmierern vermisst wird. In Abschnitt 4.2.6 wird gezeigt, wie durch TeaJay-Sprachmittel eine saubere Zählschleife nachgerüstet werden kann.

3.4.7.7. break- und continue-Statement

Das break- bzw. continue-Statement

"break" ";" | "continue" ";"

darf nur innerhalb von Schleifen vorkommen.

break: Das break-Statement beendet die Ausführung der lexikalisch nächsten Schleife und springt sofort an ihr Ende. Dies wird am besten an einem Beispiel verdeutlicht:

```
1  Number i = 0;
2  while (i < 5) {
3    Number j = 0;
4    while (true) {
5      if (j > 5) {
6        break; //beendet while(true)
7      }
8      j += 1;
9    }
10   //break springt hierhin
11   j += 1;
12 }
```

continue: Das continue-Statement beendet den aktuellen Durchlauf der lexikalisch nächsten Schleife und springt direkt zur Überprüfung der Schleifenbedingung. Zu beachten ist, dass im Falle der for-Schleife der zu Grunde liegende Iterator bei der Benutzung von continue inkrementiert wird:

```
1  List<String> lst = ...;
2  for (String str : lst) {
3    if (str == "a") {
4      continue; //beendet diesen Schleifendurchlauf, "a" wird nicht ausgegeben
5    }
6    IO.println(str);
7  }
```

break und continue existieren in TeaJay vor allem aus pragmatischen Gründen: Sie haben sich in den meisten imperativen Programmiersprachen durchgesetzt. Daher kann es durchaus sinnvoll sein, Anfängern den Umgang mit diesen Befehlen zu lehren. Die Benutzung eben jener kann es jedoch erschweren, Schleifeninvarianten zu erkennen und damit die Lesbarkeit des Quelltexts herabsetzen. Anders als Java, unterstützt TeaJay aber keine *labeled breaks* bzw. *labeled continues*.

3.4.7.8. throw-Statement

Das throw-Statement

"throw" <Expression> ";"

wird benutzt, um eine Ausnahme zu werfen. Dabei muss der Typ von <Expression> zuweisungskompatibel zu java.lang.Throwable sein. In TeaJay ist es jedoch nicht möglich direkt von Java-Klassen zu erben, daher gibt es die Typen teajay.lang.TJThrowable und teajay.lang.TJException von denen in TeaJay geerbt werden kann. Eine simple eigene Ausnahme sähe in TeaJay z.B. folgendermaßen aus:

Typdefinition:

```
1  typedef MyException extends TJException {
2    MyException();
3  }
```

Implementierung:

```
typeimpl MyExceptionImpl implements MyException {
  MyException() {
    super("Meine Nachricht");
  }
}
```

Werfen:

```
void throwTest() throws MyException {
  if (error) {
    throw new MyException();
  }
}
```

3.4.7.9. try-Statement

Das try-Statement

"try" <Block> <Catches>
<Catches> ::= <Catch> { <Catch> }
<Catch> ::= "catch" "(" <Type> <Identifier> ")" <Block>

dient der Ausnahmebehandlung. Der Block zwischen try und dem ersten catch darf nicht leer sein. Sollte innerhalb von "try" <Block> eine Ausnahme geworfen werden, kann diese durch ein <Catch> abgefangen werden. Ein catch-Block wird genau dann besucht, wenn eine Ausnahme aufgetreten ist und diese zuweisungskompatibel mit <Type> ist. Dabei wird in der Reihenfolge des Auftretens im Quelltext überprüft, ob ein catch-Fall zutrifft und auch nur der erste passende catch-Block ausgeführt. In diesem Beispiel wird der zweite catch-Block nie besucht:

```
try {
  ...
} catch (Exception ex) {
  ...
} catch (NumberFormatException ex) {
  //Nicht erreichbar
}
```

Ein catch-Fall trifft genau dann zu, wenn eine Ausnahme geworfen wurde und diese mit der im catch-Fall spezifizierten Ausnahme zuweisungskompatibel ist. Da java.lang.NumberFormatException von java.lang.Exception erbt, ist sie auch zuweisungskompatibel zu java.lang.Exception. Hier wird zuerst java.lang.Exception abgefangen und somit das zweite catch nie erreicht. Solche Fälle könnten vom Compiler erkannt werden und sollten dann zum Abbruch der Übersetzung führen. Dieses Verhalten ist jedoch im Prototypcompiler noch nicht umgesetzt.

3.4.7.10. return-Statement

Das return-Statement

"return" [<Expression>] ";"

beendet die Ausführung seiner umgebenden Methode und gibt <Expression> als dessen Rückgabewert an. Falls die Methode keinen Rückgabewert hat, ist es ein

Kompilierzeitfehler, wenn auf return ein Ausdruck folgt. return darf an jeder stelle innerhalb einer Methode vorkommen. Eine Ausnahme bilden Konstruktoren und statische Konstruktoren, welche das Benutzen von return verbieten.

3.4.7.11. Unerreichbare Statements

MB

Es gibt zwei Arten von unerreichbaren Anweisungen in einer imperativen strukturierten Programmiersprache. Die erste Art ist, wenn Anweisungen geschrieben werden, nachdem der sequenzielle Programmfluss unterbrochen wurde. In TeaJay sind es die Statements break, continue, throw und return, die den sequenziellen Ablauf innerhalb eines Blocks bzw. einer Methode unterbrechen. Nach diesen Anweisungen kann im selben Block keine weitere Anweisung folgen. Dieser Quelltext besitzt zum Beispiel unerreichbare Anweisungen:

```
1 while(i < 10 ){
2   break;
3   i +=1; //kann nicht erreicht werden
4 }
5 return;
6 Number c = 12; // kann nicht erreicht werden.
```

Die zweite Art sind unerreichbaren Anweisungen, die aufgrund von festen Programmzuständen niemals erreicht werden können. So wäre im folgenden Programm die Anweisung im else-Block unerreichbar.

```
1 if(true){
2   IO.println("Wird ausgegeben");
3 }else {
4   IO.println("Wird nie erreicht");
5 }
```

Die erste Art ist von einem Compiler feststellbar. Im Gegensatz gilt dies für die zweite Art nur bedingt. Für TeaJay wurde die Frage diskutiert, wie mit der ersten Art von unerreichbaren Anweisungen umgegangen werden soll. Hierbei wurden folgende Alternativen besprochen: Ignorieren der unerreichbaren Anweisungen und dem Entwickler eine Warnung melden, oder dies als Kompilierfehler behandeln.

Es ist festzuhalten, dass unerreichbare Anweisungen entweder durch Unwissenheit bei der Nutzung einer Programmiersprache entstehen oder versehentlich geschrieben werden. Werden die Anweisungen versehentlich geschrieben, so ist anzunehmen, dass der Entwickler einen Zweck verfolgt hat. Somit wäre das Ignorieren der Anweisungen keine Hilfe, da es sich um einen Fehler handelt. Ist die Situation durch Unwissenheit bei der Nutzung der Programmiersprache entstanden, so sollte dem Entwickler mitgeteilt werden, dass die Sprache falsch genutzt wird. Für TeaJay sind aus diesen Gründen die unerreichbaren Anweisungen der ersten Art Kompilierfehler.

Des Weiteren können die Anweisungen return und throw in bestimmten Fällen dazu führen, dass Anweisungen in einem umgebenden Block unerreichbar werden. Bei der

if-Anweisung:
wird die Ausführung im if-Block und im else-Block durch return bzw. throw

beendet, so kann nach der if-Anweisung keine weitere Anweisung folgen. Im nachfolgenden Quelltext ist ein Beispiel gezeigt:

```
1 if(true){
2   return;
3 }else{
4   return;
5 }
6 Number c = 12; // kann nicht erreicht werden
```

do-Anweisung
wird die do-Anweisung durch eine der beiden Anweisungen beendet, so kann die while-Bedingung nicht mehr geprüft werden. Hier führt die Beendigung der Ausführung durch throw bzw. return dazu, dass es zu einem Kompilierfehler bei der Angabe der while-Bedingung kommt. Im Nachfolgenden verdeutlicht ein Beispiel diesen Sachverhalt:

```
1 do{
2   return;
3 }while(true);// kann nicht erreicht werden
```

Sollte allerdings im Block der do-Anweisung eine continue- oder break-Anweisung vorkommen, so sind die Anweisungen, die nach dem do-Block folgen, erreichbar, auch wenn ein return oder throw vorkommt.

typeswitch-Anweisung
werden alle Fallböcke von der typeswitch-Anweisung durch return oder throw beendet, so sind alle Anweisungen, die der aufrufende Block nach der typeswitch-Anweisung besitzt, unerreichbar.

try-Anweisung
sind, genau wie beim typeswitch, alle Anweisungen unerreichbar, die der aufrufende Block nach der try-Anweisung besitzt, wenn der try-Block und die Catchblöcke durch return oder throw beendet werden.

Innerhalb von Schleifen können die Anweisungen break und continue in ähnlicher Weise dazu führen, dass keine weiteren Anweisungen im Schleifenblock vorkommen dürfen, beispielsweise:

```
1 while(true){
2   if( a < 10){
3     break;
4   }else{
5     continue;
6   }
7   Number c = 12; // kann nicht erreicht werden
8 }
```

Es kommt auch zu unerreichbaren Anweisungen, wenn die Fälle miteinander kombiniert werden, zum Beispiel wie folgt:

```
if(false){
  if(false){
  return;
  }else{
  return;
  }
}else{
  try{
      ...
    return;
  }catch(Throwable t){
    return;
  }
}
Number c = 10; // kann nicht erreicht werden
```

3.4.8. Ausdrücke

Die Ausdrücke in TeaJay unterscheiden sich von den in Java definierten kaum. In TeaJay werden arithmetische Ausdrücke, wie aus der Mathematik gewohnt, in Infixnotation aufgeschrieben, dabei werden Operatorvorrang und Klammerung respektiert. Ausdrücke besitzen einen eindeutigen statischen Typ der sich bei der Kompilierung ergibt. Ein Ausdruck in TeaJay ist so definiert:

<Expression> <AssignmentOperator> <Expression> |
<Expression> <InfixOperator> <Expression> |
<PrefixOperator> <Expression>|
<Primary> { <Selector> }

Durch die Teilregel <Expression> AssignmentOperator <Expression> werden Zuweisungen beschrieben. Der linke Ausdruck ist der L-Wert und der rechte Ausdruck ist der R-Wert. *R-Werte sind gewöhnlich Werte, während sich L-Werte auf den Speicherplatz beziehen [ASL+08].* Also sind Ausdrücke, die L-Werte beschreiben, insbesondere keine Methodenaufrufe oder Konstanten. Der statische Typ einer Zuweisung ist der Typ des L-Wertes. In TeaJay ist es möglich Ausdrücke dieser Form zu beschreiben:

```
Number a=1,b=1,c=1,d=1;
a += b += c += d;
IO.println(a); // Ausgabe ist 4
```

Obwohl die + =-Operatoren den gleichen Rang besitzen, werden die Operatoren des Ausdrucks nicht von links nach rechts angewendet. Dies wird deutlich wenn die Beschreibung des + =-Operators (siehe Kapitel 3.4.6) in den Ausdruck eingesetzt wird:

```
Number a=1,b=1,c=1,d=1;
a = a + ( b = b + (c = c + (d)));
IO.println(a); // Ausgabe ist 4
```

Aufgrund der Klammerung werden die Operatoren also von rechts nach links ausgewertet. Die zweite und dritte Teilregel beschreibt Ausdrücke, die einen Wert berechnen und somit R-Werte sind. Die letzte Regel beschreibt, aus welchen Operand mindestens ein Ausdruck bestehen muss. Sollte nach einem Primär-Ausdruck Selektoren folgen, so handelt es sich immer um einen R-Wert. Der Primär-Ausdruck ist durch diese Regel beschrieben:

<Literal> | “this“ | “super“
| <MethodInvocationOrVariableAccess> | <InstanceCreation>
| “(“ <Expression> “)“

Literale sind die Konstanten, die TeaJay erlaubt zu nutzen (siehe Kapitel 3.4.1). this und super sind Variablen mit einem besonderen Stellenwert in TeaJay. Hier besitzt this den Typ des implementierten Typs, wodurch sich ergibt, dass über this keine Zugriffe auf private Methoden oder Felder möglich sind. Es ist ein technisches Problem aufgrund wie Methodenaufrufe in TeaJay implementiert sind, dass mit this nur auf die öffentlichen Methoden zugegriffen werden kann. super besitzt den Typ des Supertyps des implementierten Typs und ermöglicht somit den Zugriff auf diese Schnittstelle. Die Methodenaufrufe und Variablenzugriffe werden durch die Regel beschrieben:

<Identifier> { “.“ <Identifier> } [(<AccessOperators> | <InvokeRest>)]
<AccessOperators>::= “[“ <Expression> “]“ { “[“ <Expression> “]“ }
<InvokeRest>::= [<MethodGenericParam>] <Arguments>
<MethodGenericParam>::=“.“ “<“ <ParamType> “>“ <Identifier>

Wird nur ein Bezeichner angegeben, so handelt es sich um den Zugriff auf eine Lokale- oder Feldvariable. Dieser kann sowohl ein L-Wert wie auch ein R-Wert sein. Der statische Typ dieses Ausdrucks ist der deklarierte Typ der Variable. Besitzt dieser die Instanzmethode operator_get oder operator_set, so ist es möglich, mit []-Operator auf die Variable zuzugreifen. Existiert nur operator_get, dann kann der Ausdruck nur als R-Wert verwendet werden. Existiert umgekehrt nur operator_set, kann der Ausdruck nur als L-Wert Verwendung finden. In TeaJay ist es aus Kompatibilitätsgründen möglich, auf öffentliche Felder von Objekten zuzugreifen. Statische Methoden können niemals über eine Instanz eines Typs aufgerufen werden. Des Weiteren kann ein Aufruf einer privaten statischen Methode nicht über den Typnamen aber über den Namen der Implementierung geschehen. Der Aufruf über den Namen des Typs oder der Implementierung ist notwendig, wenn eine generisch parametrisierbare statische Methode aufgerufen wird. Der Aufruf einer solchen Methode wird, wie in Java, auch über den Namen des Typs, gefolgt von einem Punkt und der konkreten Parametriesierung der Methode, ermöglicht. Das bedeutet, bei einem Typ wie

```
1 typedef Test {
2   static <X> void test(X x);
3 }
```

wird seine Methode test nicht so Test<**Number**>.test(10) sondern so Test.<↩
↪**Number**>test(10) aufgerufen. Alle Methodenaufrufe sind R-Werte und haben den definierten Rückgabetyp der Methode, als statischen Typ. Methoden die gene-

risch parametrisierbar sind oder die einen generischen Parameter als Rückgabetyp besitzen, wie z.B. dieser Typ

```
typedef TestGeneric<T extends String> {
  T genericTest();
  static <X extends String> void test(X x);
}
```

haben als statischen Typ des Methodenaufrufs die angegeben Typschranke, in diesem Fall String. Die übergebenen Argumente eines Methodenaufrufs werden durch diese Regel definiert:

<Arguments>::="(" [<Expression> { "," <Expression> }] ")"

Das heißt, es sind alle Ausdrücke erlaubt, dabei ist es unerheblich, ob der Ausdruck ein R-Wert oder ein L-Wert ist, denn die als Argumente übergebenen Ausdrücke werden immer als R-Wert interpretiert. Die Ausdrücke werden dabei von links nach rechts ausgewertet. Dabei wird anhand der ermittelten statischen Typen und des Methodennamens die aufzurufende Methode gesucht (siehe Kapitel 5.3.12). Instanzen von Typdefinitionen werden in TeaJay wie in Java mit new erzeugt. Die Instanzierung hat zur Folge, dass eine Implementierung gesucht wird und ein passender Konstruktor aufgerufen wird. Die folgende Regel beschreibt die Instanzierung von Typdefinitionen:

"new" <Type> "(" <Arguments> ")"

Wird dabei die erzeugte Instanz einer Variablen zugewiesen. So wird geprüft, ob der Typ der Variablen zu dem zu erzeugenden Typ zuweisungskompatibel ist. Des Weiteren ist der statische Typ dieses Ausdrucks der zu erzeugende Typ. Die aus Java 7 eingeführte Typinferenz existiert allerdings in TeaJay noch nicht. Jeder Primärausdruck besitzt einen statischen Typ. Auf seine Schnittstelle lässt sich mit der Selector-Regel zugreifen:

"." <Identifier> | <Arguments>

Auch jeder Selektor hat einen eindeutigen statischen Typ und auf seine Schnittstelle lässt sich wiederum zugreifen. Allerdings lässt sich die Selector-Regel nicht auf alle Typen gleichermaßen anwenden:

- Ist der Primärausdruck this oder super, kann die Arguments-Regel nur folgen, wenn dies in einem Konstruktor als erste Anweisung geschieht. Allerdings gilt für this die Ausnahme, dass die Arguments-Regel folgen darf, wenn dies in einem Closure geschieht. Dies ermöglicht den rekursiven Aufruf von Closures.

- Ist der Primärausdruck eine Zahlkonstante, so hat der Ausdruck den Typ Number. Allerdings ist es nicht möglich auf die Schnittstelle von Number über den Selector zuzugreifen. Dies liegt darin begründet, dass eine Zahl in TeaJay auch eine Fließkommazahl sein kann und es zu Realisierungsschwierigkeiten kommen kann, eine Zahl eindeutig zu identifizieren. Wie z.B. hier:

```
IO.println(2.toString());
```

Durch die technische Umsetzung wird versucht 2.toString als Zahl zu interpretieren und dies führt zu einem Fehler. Es ist aber möglich, durch das einklammern der 2 auf die Schnittstelle von Number zuzugreifen, z.B so:

```
IO.println( (2).toString());
```

- Ist der Typ des Ausdrucks ein Closure, lässt sich dieser über die Arguments-Regel aufrufen.

3.5. Kompatibilität zu Java

MK

TeaJay ermöglicht es, in begrenztem Umfang, mit Java-Typen zu kommunizieren. Dies hatte jedoch keine Priorität bei der Entwicklung des Compilers und daher ist es recht umständlich mit Java-Typen zu kommunizieren. Die Möglichkeit mit Java-Typen zu kommunizieren soll es Entwicklern erlauben TeaJay-Bibliotheken, unter Zuhilfenahme der bereits existierenden Java-Codebasis, zu entwickeln. Dieser Sprachaspekt sollte weniger Teil der Lehre sein, sondern dient eher dem Pragmatismus.

Bei der Nutzung von Java-Typen gelten folgende Einschränkungen:

- TeaJay-Typen können nicht von Java-Klassen erben.
- TeaJay-Typen können Java-Interfaces implementieren, falls alle Methoden nur Referenztypen und keine Array-Typen in ihrer Signatur haben.
- TeaJay-Interfaces können nur unter o.g. Bedingungen von Java-Interfaces erben.
- Verschachtelte Klassen werden nicht vom Compiler erkannt.
- Beim Umgang mit Array-Typen wird die statische Typsicherheit nur eingeschränkt geprüft.
- Es können keine *raw types* genutzt werden.

Java-Klassen können auf dieselbe Weise wie TeaJay-Typen instanziiert werden:

```
//Achtung: import java.util.*; überschreibt teajay.lang.List mit java.util.List
import java.util.ArrayList;
//Code:
ArrayList<String> jlst = new ArrayList<String>();
jlst.add("some string");
IO.println(jlst);
```

Komplizierter wird es, wenn mit Java-Primitiven umgegangen werden muss. Im Paket teajay.lang.java finden sich Werkzeuge für den Umgang mit primitiven Java-Typen und Arrays. Folgendes Beispiel zeigt, wie es möglich ist mit Java-Primitiven in TeaJay umzugehen:

```
import teajay.lang.java.*;
import java.util.ArrayList;
//Code:
ArrayList<String> jlst = new ArrayList<String>(JavaPrimitives.toInt(16));
jlst.add("");
Number size = JavaPrimitives.toNumber(jlst.size());
IO.println(size); // Ausgabe: 1
```

Die Methode JavaPrimitives.toInt(Number) hat als Rückgabewert den primitiven Typ int. In TeaJay ist es nicht möglich, Java-Primitive auszudrücken. Daher kann man den Rückgabewert von JavaPrimitives.toInt(Number) nur direkt an eine andere Java-Methode weiterleiten. Die Typsicherheit wird in diesem Fall aber garantiert. Es wird jedoch keine der aus Java bekannten impliziten Typkonvertierungen durchgeführt (z.B. von short nach int). Im Fall von Methoden, die boolean als Rückgabewert haben, wird immer eine implizite Typkonvertierung des Rückgabewerts zu Boolean durchgeführt. Für den Umgang mit Java-Arrays wurde ein Artefakt in die Sprache eingebaut: Der Cast-Operator unterstützt das Casten zu Java-Arrays. Folgendes Beispiel zeigt den Umgang mit primitiven Java-Arrays in TeaJay:

```
1  package implementations;
2  import java.io.*;
3  import teajay.lang.java.*;
4  /* Schreibt den Inhalt der Standardeingabe oder file auf die Standardausgabe. */
5  typeimpl JavaIOExampleImpl implements javaio.JavaIOExample {
6    String filename;
7
8    JavaIOExampleImpl(String file) {
9      filename = file;
10   }
11   JavaIOExampleImpl() {
12   }
13   void run() {
14     try {
15       InputStream in;
16       if (filename != null) {
17         in = new FileInputStream(filename);
18       } else {
19         in = System.in;
20       }
21       ByteArray<Number> buffer = new ByteArray<Number>(1024);
22       Number read = JavaPrimitives.toNumber(in.read((byte[]) buffer.getJavaArray()));
23       while (read > -1) { //Prüfe auf EOF
24         System.out.write((byte[]) buffer.getJavaArray(),
25           JavaPrimitives.toInt(0), JavaPrimitives.toInt(read));
26         read = JavaPrimitives.toNumber(in.read((byte[]) buffer.getJavaArray()));
27       }
28       in.close();
29     } catch (IOException ex) {
30       IO.println1nErr("Fehler: " + ex.getMessage());
31     }
32   }
33 }
```

Quellcode 3.10: JavaIOExampleImpl.tj

ByteArray<Number> steht für ein eindimensionales Array. Ein zweidimensionales Array der Größe 5×5 kann man wie folgt erzeugen:

```
1  ByteArray<ByteArray<Number>> array2D = new ByteArray<ByteArray<Number>>(5, 5);
```

Die Typargumente von ByteArray werden dabei vom Compiler noch nicht auf Plausibilität geprüft und müssen im Fall von primitiven Arrays immer auf Number enden. Im Paket teajay.lang.java gibt es weitere Klassen, die es ermöglichen mit anderen Arten von Java-Arrays zu arbeiten (z.B. ReferenceArray<T> für den Umgang mit Arrays von Referenztypen). Möchte man ein Java-Array in einen der Wrapper-Typen aus teajay.lang.java verpacken, ist dies wie folgt möglich:

```
1  CharArray<Number> abs = new CharArray<Number>("abc".toCharArray());
```

Dabei ist zu beachten, dass auch hier die statische Typsicherheit nicht garantiert bzw. überprüft wird: Der entsprechende Konstruktor von CharArray hat die Signatur CharArray(Object obj) und prüft zur Laufzeit, ob es sich bei obj um ein char-Array beliebiger Dimension handelt. Aus TeaJay heraus ist es mit der aus Java bekannten Syntax auch möglich, auf Felder von Java-Klassen lesend sowie schreibend zuzugreifen. Eine weitere Maßnahme, um die Kommunikation mit Java-Klassen zu vereinfa-

chen ist, dass sich Object.equals(Object) für TeaJay-Typen genauso verhält wie der Vergleichsoperator.

4. Exkurs: TeaJay anwenden

MK

Dieses Kapitel soll Anregungen zur Anwendung von TeaJay in Lehre und Praxis geben, um so TeaJay dem Leser nahezubringen. Zudem wird auf einige Aspekte der Entwicklung in TeaJay hingewiesen und es werden Eigenarten des Typsystems diskutiert. Dies geschieht vor allem anhand von Beispielen.

4.1. TeaJay in der Lehre

Die Programmiersprache TeaJay soll dazu dienen den Umgang mit ADTs zu lehren, indem die Sprache das Geheimnisprinzip forciert. Wie in Kapitel 3 bereits erwähnt, bietet TeaJay dem Entwickler keine Möglichkeit, Informationen über die Interna eines Datentyps zu erfahren und erzwingt damit eine Kapselung von Daten. Zudem unterstützt TeaJay Closures, um das Lehren funktionaler Programmierung zu erleichtern. In diesem Kapitel werden einige Vorschläge zum Einsatz von TeaJay in der Lehre gemacht.

4.1.1. Erste Schritte in TeaJay

Im Folgenden werden einige simple TeaJay-Programme vorgestellt, welche dazu geeignet sind, die wichtigsten Sprachelemente von TeaJay zu demonstrieren. Dies soll dazu dienen, einen Lernenden in die Lage zu versetzen, eigene lauffähige Programme in TeaJay zu erstellen.

4.1.1.1. Das erste TeaJay-Programm

Aufgabe ist es, ein Programm zu entwickeln, welches zwei Zahlen als Kommandozeilenparameter akzeptiert und deren Summe ausgibt. Ein Aufgabensteller kann hier z.B. die Typbeschreibung bereits mitliefern und dem Lernenden das Entwickeln einer Implementierung überlassen. Eine Typbeschreibung für ein solches Programm könnte wie folgt aussehen:

```
1  package simpleadder;
2  /*
3   * Addierer: Erlaubt das Addieren zweier Zahlen.
4   */
5  public typedef SimpleAdder {
6    /*
7     * Erwartet zwei als String codierte Zahlen.
8     */
9    SimpleAdder(String a, String b) throws NumberFormatException;
10   /*
11    * Führt die Addition a + b aus und gibt das Ergebnis zurück.
12    */
13   Number add();
14 }
```

Quellcode 4.1: SimpleAdder.tj

Da der Typ simpleadder.SimpleAdder ausführbar sein soll, darf der Konstruktor nur Strings als Parameter akzeptieren. Vorausgesetzt SimpleAdder.tj befindet sich im Verzeichnis simpleadder (relativ zum aktuellen Arbeitsverzeichnis), lässt sich der Typ mit folgenden Befehlen übersetzen und ausführen:

```
1  > tjc simpleadder/SimpleAdder.tj
2  > tj simpleadder.SimpleAdder add 3 6.5
3  Please state an implementation for simpleadder.SimpleAdder:
```

Man beachte dabei, dass man den Namen der Methode, welche als Einstiegspunkt dienen soll, angeben muss (in diesem Fall die Methode add()). Sofern noch keine Implementierung vorliegt, wird die Laufzeitumgebung, wie im Beispiel gezeigt, nun nach dem Namen einer Implementierung fragen. Dabei muss sich eine hier angegebene Implementierung bereits im Suchpfad von TeaJay befinden. Eine Implementierung könnte wie folgt aussehen:

```
1  package implementations;
2
3  import simpleadder.SimpleAdder;
4
5  typeimpl SimpleAdderImpl implements SimpleAdder {
6    Number a;
7    Number b;
8
9    SimpleAdderImpl(String a1, String b1) throws NumberFormatException {
10     a = Number.parseNumber(a1);
11     b = Number.parseNumber(b1);
12   }
13
14   Number add() {
15     return a + b;
16   }
17 }
```

Quellcode 4.2: SimpleAdderImpl.tj

Wenn sich die Implementierung (und die vom Compiler generierte .tjt-Datei) im Suchpfad von tj befindet, ergibt ein erneutes Ausführen zum Beispiel das folgende Resultat:

```
1  > tjc implementations/SimpleAdderImpl.tj
2  > tj simpleadder.SimpleAdder add 3 6.5
3  9.5
4  > tj simpleadder.SimpleAdder add 4/5 2/3
5  1.46666666666666666666666666666667
```

Man beachte, dass beim zweiten Beispiel die interne Darstellung des Ergebnisses exakt $\frac{22}{15}$ ist. Die Methode Number.toString() arbeitet für echt rationale Zahlen mit einer Genauigkeit von 34 Dezimalziffern[1]. Falls die als Einstiegspunkt angegebene Methode einen Rückgabewert hat, wird die Laufzeitumgebung, wie im Beispiel zu erkennen, dessen String-Repräsentation beim Herunterfahren der JVM auf der Konsole ausgeben. Die Kommandozeilenparameter werden im Konstruktor SimpleAdderImpl(String, String) übergeben.

An diesem Beispiel lässt sich einerseits erlernen wie einfache Kommandozeilenparameter in TeaJay verarbeitet werden können und andererseits wie ein TeaJay Programm ausgeführt werden kann. Ein TeaJay-Typ ist ausführbar, wenn er eine parameterlose Methode besitzt und eine der folgenden Bedingungen zutrifft:

- es gibt einen leeren Konstruktor

[1]Entspricht laut [MC13] dem IEEE 754R Decimal128 Standard.

- es gibt einen Konstruktor, der nur Strings als Argumente erwartet
- es gibt einen Konstruktor, der genau eine List<String> als Argument erwartet

Wie auch bei Java-Klassen, ist es in TeaJay wichtig, dass sich die class-Dateien in einer Verzeichnisstruktur befinden, die den Paketnamen wiederspiegelt. Der TeaJay-Compiler legt diese Verzeichnisstruktur relativ zum Ausgabepfad (-d) selbstständig an. Es wird empfohlen, auch Quellcodedateien in einer der Paketstruktur entsprechenden Verzeichnisstruktur anzulegen. Damit wird der Compiler in die Lage versetzt, benötigte Typen automatisch zu finden, auch wenn sie nur als Quellcode vorliegen.

4.1.1.2. FancyAdder: Eine Weiterentwicklung des SimpleAdder

Der SimpleAdder aus 4.1.1.1 lässt sich zu einem Summierer erweitern, welcher beliebig viele Zahlen addieren kann. Dabei wird TeaJays einfachste Schleifenform eingeführt: Die for-Schleife. In TeaJay iteriert eine for-Schleife über alle Elemente eines iterierbaren Datentyps (z.B. List). Um iterierbar zu sein, muss ein Datentyp das interface teajay.lang.TJIterable oder java.lang.Iterable implementieren. Wie in 4.1.1.1 wird zuerst die Typdefinition angegeben:

```
1  package fancyadder;
2  /*
3   * Summierer: Erlaubt das Summieren beliebig vieler Zahlen.
4   */
5  public typedef FancyAdder {
6    /*
7     * Erwartet eine Liste von Zahlen (als String codiert).
8     */
9    FancyAdder(List<String> input) throws NumberFormatException;
10   /*
11    * Summiert alle Zahlen aus input auf und gibt das exakte Ergebnis auf der Konsole aus.
12    */
13   void sum();
14 }
```

Quellcode 4.3: FancyAdder.tj

Um die Kontrolle über das Ausgabeformat zu haben, hat sum() keinen Rückgabewert sondern die Nebenwirkung, das Ergebnis auf der Konsole auszugeben. Der Konstruktor von FancyAdder erwartet eine List<String>. Das Programm akzeptiert daher beliebig viele Kommandozeilenparameter. Eine Implementierung von FancyAdder könnte wie folgt aussehen:

```
1  package implementations;
2
3  import fancyadder.FancyAdder;
4
5  typeimpl FancyAdderImpl implements FancyAdder {
6    List<Number> numbers; //Eine Liste von Zahlen
7
8    FancyAdderImpl(List<String> vals) throws NumberFormatException {
9      numbers = new List<Number>();
10     for (String val : vals) {
11       numbers.add(Number.parseNumber(val));
12     }
13   }
14
15   void sum() {
16     Number result = 0; //Die leere Summe ist 0
17     for (Number num : numbers) {
18       result += num; //<=> result = result + num;
19     }
20     IO.println(result.toFractionString()); //Ausgabe als Bruch
21   }
22 }
```

Quellcode 4.4: FancyAdderImpl.tj

Für Lernende, die bereits Vorkenntnisse z.B. in Java oder Python haben, wird die gezeigte Form der for-Schleife nichts Überraschendes haben. Anzumerken ist, dass ähnlich wie in Java, während des Iterierens über eine Sammlung diese nur über den Iterator verändert werden sollte (dazu dient TJIterator.remove()). Da eine for-Schleife keinen Zugriff auf ihren Iterator erlaubt, sollte innerhalb einer for-Schleife die zugrunde liegende Sammlung nie verändert werden.

In diesem Beispiel wurde bereits der parametrisierte Typ List<String> benutzt. Ein Lehrender sollte hier nur kurz auf diese Tatsache hinweisen und List <String> zunächst wie einen speziellen Typen behandeln. Das generische Typsystem von TeaJay sollte aufgrund seiner doch recht hohen Komplexität dem Lernenden schrittweise nahe gebracht werden.

4.1.1.3. Der ADT Stack

In TeaJay ist jedes Programm ein ADT. Daher waren die beiden vorangegangenen TeaJay-Programme auch nur Implementierungen eines ADTs. Nur die Operationen (Methoden und Konstruktoren) und Sorten (Signaturen eben jener) werden für den Compiler sichtbar angegeben, während die Semantik der Schnittstelle informell in Kommentaren angegeben werden muss.[2]

Bei der Besprechung des vorigen Beispielprogramms ist bereits der Begriff „parametrisierter Typ“ gefallen. Am folgenden Beispiel kann ein Lehrender nun zeigen, wie man generische Typen, welche parametrisierbar sind, selber definieren kann:

```
1  package stack;
2  /**
3   * ADT: Stack (Last In - First Out)
4   */
5  public typedef Stack<T> {
6    /**
7     * Erzeugt einen leeren Stack.
8     */
9    Stack();
10   /**
11    * Entfernt das oberste Element des Stacks.
12    * Wirft eine IllegalStateException falls der Stack leer ist.
13    */
14   T pop() throws IllegalStateException;
15   /**
16    * Legt ein Element auf den Stack.
17    */
18   void push(T a);
19   /**
20    * Gibt das oberste Element des Stacks zurück, ohne es zu entfernen.
21    * Wirft eine IllegalStateException falls der Stack leer ist.
22    */
23   T peek() throws IllegalStateException;
24   /**
25    * Gibt die Anzahl der Elemente, die sich auf dem Stack befinden, zurück.
26    */
27   Number size();
28 }
```

Quellcode 4.5: Stack.tj

Die Typvariable T steht in diesem Fall für einen beliebigen Typ und hat damit die Typgrenze Object. Für dieses simple Beispiel kann man Parametrisierung noch als eine Art von Textersetzung erklären (T wird durch einen konkreten Typ ersetzt), jedoch wird das der Sache nicht völlig gerecht. Früher oder später sollte den Lernenden erklärt werden, dass es sich bei der Definition von Stack<T > eigentlich um die

[2] In Kapitel 6 werden Möglichkeiten diskutiert, die Semantik formal prüfbar oder mithilfe von Testtreibern anzugeben bzw. automatisch zu testen.

Definition einer Menge von Typen handelt und T eine Variable ist, welche Typinformationen enthalten kann. Für jede Ausprägung der Typvariablen T steht Stack<T> für einen jeweils anderen Typ. Diese Typen existieren jedoch nur auf Sprachebene.

Zur Implementierung von Stack könnte man den in TeaJay eingebauten Datentyp List nutzen oder über Referenzen eine Verkettung aufbauen. Da die zweite Möglichkeit für die Lehre interessant ist, wurde diese gewählt. Dazu wird zuerst ein neuer generischer ADT namens StackElem definiert:

```
1  package implementations;
2  /**
3   * ADT: StackElem
4   * Beispiel eines "package private" Typen.
5   */
6  typedef StackElem<T> {
7    /**
8     * Erzeugt ein neues Stackelement mit dem Wert value und next als nächstes Element.
9     * next sowie value dürfen null sein, wobei next == null bedeutet dass dies das letzte ↩
          ↪Element des Stacks ist.
10    */
11   StackElem(T value, StackElem<T> next);
12   /**
13    * Gibt den Wert dieses Stackelements zurück.
14    */
15   T getValue();
16   /**
17    * Gibt das nächste Element zurück. (null falls es keines mehr gibt)
18    */
19   StackElem<T> getNext();
20 }
```

Quellcode 4.6: StackElem.tj

Da der Typ StackElem ein Implementierungsdetail von Stack ist, befindet er sich im Paket implementations und wurde als package private gekennzeichnet (durch Weglassen des Schlüsselwortes public). Ein Klient braucht kein Wissen darüber, dass eine Implementierung von Stack einen Typ StackElem benötigt. Die Implementierung von StackElem ist trivial und wird daher nicht angegeben. Unter Benutzung von StackElem kann man nun einen Stack wie folgt implementieren:

```
1  package implementations;
2  import stack.Stack;
3  /**
4   * Implementiert den ADT stack.Stack durch Verkettung von Elementen.
5   */
6  typeimpl StackImpl<T> implements Stack<T> {
7    StackElem<T> top;
8
9    StackImpl() { top = null; }
10   T pop() throws IllegalStateException {
11     if (top == null) {
12       throw new IllegalStateException("empty stack"); //Bricht den Ablauf der Methode sofort ab
13     }
14     T value = top.getValue();
15     top = top.getNext();
16     return value;
17   }
18   void push(T a) {
19     top = new StackElem<T>(a, top);
20   }
21   T peek() throws IllegalStateException {
22     if (top == null) {
23       throw new IllegalStateException("empty stack");
24     }
25     return top.getValue();
26   }
27   Number size() {
28     StackElem<T> elem = top;
29     if (elem == null) {
30       return 0; //Bricht den Ablauf der Methode sofort ab
31     }
32     Number size = 1;
33     while (elem.getNext() != null) {
34       size += 1;
35       elem = elem.getNext();
36     }
37     return size;
38   }
39 }
```

Quellcode 4.7: StackImpl.tj

In diesem Beispiel werden Ausnahmen zur Fehlermeldung, die Kontrollstrukturen if sowie while und der Umgang mit dem Typ null eingeführt.

Der TeaJay-Compiler zwingt den Programmierer momentan nicht, Ausnahmen zu behandeln. Dies ist jedoch dem Prototypstatus des Compilers geschuldet, da überprüfte Ausnahmen[3] nach Meinung der Entwickler von TeaJay in einer Lehrsprache sinnvoll sind.

4.1.1.4. Benutzen von selbstdefinierten ADTs in TeaJay

Im Folgenden wird ein kleines Programm vorgestellt, welches mathematische Ausdrücke in Postfixnotation auswertet. Dafür wird der in 4.1.1.3 entwickelte Stack genutzt. Zuerst die Typdefinition von PostfixEvaluator:

```
1  package postfix;
2  /**
3   * PostfixEvaluator: Erlaubt das Auswerten mathematischer Postfixausdrücke.
4   * Unterstützte Operatoren: +, *, -, /
5   */
6  public typedef PostfixEvaluator {
7    /**
8     * Akzeptiert einen in Token zerlegten Postfixausdruck und wertet ihn aus.
9     */
10   static Number eval(List<String> expr) throws NumberFormatException, IllegalArgumentException;
11 }
```

Quellcode 4.8: PostfixEvaluator.tj

PostfixEvaluator besitzt nur eine statische Methode und keinen Konstruktor. Aus TeaJays Sicht ist er dennoch ein ADT, denn statische Methoden werden mit zur öffentlichen Schnittstelle des Typs gezählt. PostfixEvaluator bietet in seiner öffentlichen Schnittstelle aber keinen Konstruktor an und kann folglich nicht instanziiert werden. In der Implementierung kann man nun sehen wie der ADT Stack verwendet wird:

```
1  package implementations;
2
3  import postfix.PostfixEvaluator;
4  import stack.Stack;
5
6  typeimpl PostfixEvaluatorImpl implements PostfixEvaluator {
7
8    static Number eval(List<String> expr) throws NumberFormatException, IllegalArgumentException ↩
       ↪{
9      //Hier wird die Laufzeitumgebung aufgefordert eine bel. Implementierung von stack.Stack zu ↩
         ↪laden
10     Stack<Number> operandStack = new Stack<Number>();
11     for (String str : expr) {
12       if (isBinaryOp(str)) {
13         applyBinaryOp(operandStack, str);
14       } else {
15         operandStack.push(Number.parseNumber(str));
16       }
17     }
18     if (operandStack.size() > 1) {
19       throw new IllegalArgumentException("stack overflow");
20     } else if (operandStack.size() <= 0) {
21       throw new IllegalArgumentException("stack underflow");
22     }
23     return operandStack.pop();
24   }
25
26   private static Boolean isBinaryOp(String str) {
27     return str == "+" || str == "*" || str == "-" || str == "/";
28   }
29
30   private static void applyBinaryOp(Stack<Number> operandStack, String op) throws ↩
       ↪IllegalArgumentException {
31     if (operandStack.size() < 2) {
32       throw new IllegalArgumentException("stack underflow");
33     }
34     Number b = operandStack.pop();
```

[3] Überprüfte Ausnahmen (englisch: *checked exceptions*) sind Ausnahmen, bei denen der Compiler sicherstellt, dass Code für die Ausnahmebehandlung existiert.

```
35      Number a = operandStack.pop();
36      if (op == "+") {
37        operandStack.push(a + b);
38      } else if (op == "-") {
39        operandStack.push(a - b);
40      } else if (op == "*") {
41        operandStack.push(a * b);
42      } else if (op == "/") {
43        operandStack.push(a / b);
44      } else {
45        throw new AssertionError("no such operation: " + op);
46      }
47    }
48  }
```

Quellcode 4.9: PostfixEvaluatorImpl.tj

Hier kann man gut erkennen, dass der Klient von Stack keinerlei Informationen braucht, die über die öffentliche Schnittstelle hinausgehen, um Stack zu benutzen. Genauso würde es sich auch mit einem Klienten von PostfixEvaluator verhalten. Für den ADT PostfixEvaluator wird nun noch ein Testtreiber erstellt, der es erlaubt, Postfixausdrücke auf der Kommandozeile zu übergeben und auswerten zu lassen:

```
1   package postfix;
2   /**
3    * Testet den PostfixEvaluator indem er ein Kommandozeileninterface zur Verfügung stellt.
4    */
5   public typedef PostfixEvaluatorTest {
6     /**
7      * Akzeptiert eine Liste von Postfixausdrücken.
8      */
9     PostfixEvaluatorTest(List<String> args);
10    /**
11     * Wertet alle Ausdrücke aus und zeigt das Ergebnis an.
12     */
13    void eval();
14  }
```

Quellcode 4.10: PostfixEvaluatorTest.tj

```
1   package postfix;
2   import java.util.StringTokenizer;
3   typeimpl PostfixEvaluatorTestImpl implements PostfixEvaluatorTest {
4     List<String> expressions;
5
6     PostfixEvaluatorTestImpl(List<String> args) {
7       expressions = args;
8     }
9     void eval() {
10      for (String expr : expressions) {
11        try {
12          StringTokenizer tok = new StringTokenizer(expr, " ");
13          List<String> tokenized = new List<String>();
14          while (tok.hasMoreElements()) {
15            tokenized.add(tok.nextToken());
16          }
17          IO.println(expr + "  =  " + PostfixEvaluator.eval(tokenized));
18        } catch (NumberFormatException ex) {
19          IO.println(expr + " << malformed postfix expression: " + ex.getMessage() + ">>");
20        } catch (IllegalArgumentException ex) {
21          IO.println(expr + " << malformed postfix expression: " + ex.getMessage() + ">>");
22        }
23      }
24    }
25  }
```

Quellcode 4.11: PostfixEvaluatorTestImpl.tj

Einen solchen Testtreiber kann ein ADT-Spezifizierer mitliefern und z.B. noch um Tests erweitern, die bestimmte Eigenschaften der Implementierung testen. Möchte ein Lehrender verhindern, dass der Testtreiber von den Lernenden eingesetzt wird, reicht es die Typdefinition von PostfixEvalutator vorzugeben, um damit eine zum Testtreiber kompatible Signatur zu erzwingen. An diesem Beispiel wird auch gezeigt, wie Ausnahmen in TeaJay behandelt werden können.

4.1.2. Entwerfen von Typen mit TeaJay

Eine Besonderheit von TeaJay gegenüber Java ist, dass TeaJay Operatorüberladung unterstützt. Diese Tatsache gibt einem Schnittstellendesigner mehr Möglichkeiten zur Ausgestaltung einer Schnittstelle eines selbstdefinierten Typs. Operatoren sind in den Köpfen der meisten Entwickler jedoch mit einer bestimmten Bedeutung belegt und daher ist es eine besondere Herausforderung für einen Schnittstellenentwickler den Operatoren eine intuitive Semantik zu geben. Im Folgenden werden mehrere Datentypen vorgestellt und auf den Aspekt der Operatorüberladung eingegangen.

Im vorigen Abschnitt sind die Aufgabenstellungen vor allem auf das Implementieren von Datentypen ausgerichtet. Ein Lernender sollte nun in der Lage sein, die grundlegenden Sprachelemente von TeaJay zu beherrschen und ein Lehrender sollte nun auch das Entwerfen der Schnittstellen in die Hand der Lernenden legen. Der Autor ist der Meinung, dass das Entwerfen von Schnittstellen bereits zu einem sehr frühen Zeitpunkt gelehrt werden sollte, damit sich keine Nachlässigkeit in diesem Zusammenhang einschleift.

4.1.2.1. Der Typ Complex

Aufgabe ist es, eine Schnittstelle für den Typ Complex, welcher eine komplexe Zahl darstellen soll, zu entwerfen. Zudem sollen alle, für diesen Typ sinnvollen Operatoren, in einer sinnvollen Weise überladen werden. Was genau "sinnvoll" in diesem Zusammenhang bedeutet, ist subjektiv. Zum Beispiel hält der Autor es für sinnvoll, die arithmetischen Operatoren auch für den Typ Number zu überladen. Die Überladung der Vergleichsoperatoren jedoch hält er für wenig sinnvoll, da der Körper der komplexen Zahlen nicht angeordnet ist. Ein Entwickler könnte jedoch auf die Idee kommen, die Vergleichsoperatoren über den Betrag der komplexen Zahl zu realisieren (und dies gut zu dokumentieren). Es folgt ein Vorschlag für die Definition des Typs Complex:

```
1  package complex;
2
3  public typedef Complex {
4    Complex(Number real, Number imag);
5    Complex(Number real);
6
7    static Complex operator_add(Complex a, Complex b);
8    static Complex operator_add(Number a, Complex b);
9    static Complex operator_add(Complex a, Number b);
10
11   static Complex operator_mult(Complex a, Complex b);
12   static Complex operator_mult(Number a, Complex b);
13   static Complex operator_mult(Complex a, Number b);
14
15   static Complex operator_div(Complex a, Complex b);
16   static Complex operator_div(Number a, Complex b);
17   static Complex operator_div(Complex a, Number b);
18
19   static Complex operator_sub(Complex a, Complex b);
20   static Complex operator_sub(Number a, Complex b);
21   static Complex operator_sub(Complex a, Number b);
22
23   static Complex operator_sub(Complex a);
24
25   Boolean operator_eq(Object obj);
26
27   Complex add(Complex other);
28   Complex mult(Complex other);
29   Complex div(Complex other);
30   Complex sub(Complex other);
31   /* komplexe konjugation */
32   Complex conjugate();
33
34   /* Realteil */
35   Number re();
36   /* Imaginärteil */
```

```
37    Number im();
38    /* Absolutbetrag */
39    Number abs(Number precision);
40    /* Übersetzt einen String in eine Komplexe Zahl */
41    static Complex parseComplex(String str) throws NumberFormatException;
42
43    String toString();
44    String toFractionString();
45  }
```

Quellcode 4.12: Complex.tj

Bei dieser Typdefinition sollten drei Dinge besonders auffallen:

- Zur Überladung der arithmetischen Operatoren für Number wurden immer jeweils zwei Methoden definiert:

```
1     static Complex operator_...(Number a, Complex b);
2     static Complex operator_...(Complex a, Number b);
```

  Dies stellt sicher, dass die Operatoren wie erwartet funktionieren, wie dieses Negativbeispiel zeigt:

```
1     typedef Complex {
2       ...
3       //Angenommen Complex überschreibt * nur mit diesen Methoden
4       static Complex operator_mult(Complex a, Complex b);
5       static Complex operator_mult(Number a, Complex b);
6       ...
7     }
8     //Code:
9     Complex a = new Complex(4, 5);
10    Number b = 7;
11
12    b * a; //OK ruft operator_mult(Number, Complex) auf
13    a * b; //Fehler, da keine Methode operator_mult(Complex, Number) gefunden werden ↩
          ↪kann
```

  Für Complex ist dieses Verhalten nicht gewollt, jedoch kann, je nach Kontext, dieses Verhalten gewollt sein, wenn z.B. a * b gar nicht definiert wäre. Für Complex lässt sich aber a * b durchaus sinnvoll definieren und daher sollte dieses auch getan werden.

- Die Methode abs(Number), welche den Absolutbetrag berechnen soll, verlangt die Angabe einer numerischen Präzision. Das liegt daran, dass Number keine statische Genauigkeit hat und daher bei allen numerischen Operationen die Genauigkeit vom Benutzer festgelegt werden sollte. Dies muss ein Entwickler bei der Schnittstellendefinition beachten.

- Zusätzlich zu den arithmetischen Operatoren existieren die Instanzmethoden add, mult, sub und div. Das liegt zum Einen daran, dass die Operatoren nur auf die öffentliche Schnittstelle des Typs zugreifen können und zum Anderen daran, dass Operatoren nicht vererbt werden. Einer Spezialisierung von Complex würde damit essentielle Funktionalität des Supertyps fehlen.

Bei der Implementierung von Complex.operator_eq(Object) kommt ein neues Sprachelement vor: typeswitch

```
1   ...
2   Boolean operator_eq(Object other) {
3     typeswitch (other) {
4       case Complex {
5         return other.re() == this.re() && other.im() == this.im();
6       }
7       case Number {
8         return this.im() == 0 && this.re() == other;
9       }
10      default {
```

```
11          return false;
12        }
13      }
14    }
15    ...
```

Quellcode 4.13: ComplexImpl.tj

typeswitch ersetzt den in TeaJay nicht verfügbaren Java-Operator instanceof und soll ungeprüfte Typumwandlungen vermeiden helfen.

Mit der o.g. Implementierung ist folgender Ausdruck wahr:

```
1  45 == new Complex(45, 0) && new Complex(45, 0) == 45 //true
```

Dabei sei nochmals auf die Besonderheit des Vergleichsoperators, anzunehmen, dass die Gleichheitsbeziehung kommutativ ist, hingewiesen. Daher wird bei 45 == new Complex(45,0) nicht nur (45).operator_eq(new Complex(45, 0)) aufgerufen, sondern bei Bedarf auch new Complex(45, 0).operator_eq(45) und die beiden Ergebnisse Oder-Verknüpft.

4.1.2.2. Der Typ Map

Eine typische Aufgabe in einem Anfängerkurs für Informatik ist es, eine Hashtabelle zu implementieren. Der Typ Map definiert dabei nur eine Datenstruktur, die eine eindeutige Schlüssel-Wert-Beziehung herstellt. Solche Datenstrukturen werden aber häufig durch eine Hashtabelle implementiert. Es folgt die Typdefinition von Map:

```
1  package map;
2  /* Implementiert eine Indexstruktur über beliebige Schlüssel-Wertpaare. */
3  public typedef Map<K, V> interfaces TJIterable<MapEntry<K, V>> {
4    Map();
5    /* Fügt ein Element hinzu. */
6    void put(K key, V value);
7    /* Holt den zu key passenden Wert aus der Tabelle. */
8    V get(K key);
9
10   void operator_set(K key, V value);
11   V operator_get(K key);
12   /* Entfernt den Schlüssel key aus der Tabelle. */
13   V remove(K key);
14   /* Prüft, ob sich der Schlüssel key in der Tebelle befindet. */
15   Boolean contains(K key);
16   /* Gibt die Anzahl von Schlüsseln in der Tabelle zurück. */
17   Number size();
18
19   List<K> keys();
20   List<V> values();
21   List<MapEntry<K, V>> entries();
22 }
```

Quellcode 4.14: Map.tj

Der Typ Map erfüllt das interface TJIterable<MapEntry<K, V>>. Daher ist es möglich, in einer for-Schleife über ihn zu iterieren und man kann an ihm beispielhaft das Interfacekonzept von TeaJay einführen.

```
1  Map<String, Number> someMap = ...;
2  for (MapEntry<String, Number> entry : someMap) {
3    IO.println(entry.getKey() + " -> " + entry.getValue());
4  }
```

Quellcode 4.15: for-Schleife, die über Map iteriert

Im Folgenden werden nur interessante Stellen der Implementierung besprochen und auf eine vollständige Auflistung verzichtet:

iterator() :

```
1  ...
2  TJIterator<MapEntry<K, V>> iterator() {
3    return new MapIterator<K, V>(this);
4  }
5  //MapIterator
6  typedef MapIterator<K, V> interfaces TJIterator<MapEntry<K, V>> {
7    MapIterator(Map<K, V> map);
8  }
9  ...
```

Quellcode 4.16: MapImpl.tj

Hier sieht man, dass TeaJay auch manche Dinge verkompliziert. In Java würde man einfach eine anonyme Implementierung von TJIterator erzeugen. Diese hätte auch Zugriff auf Interna von MapImpl. MapIterator kann dagegen nur die öffentliche Schnittstelle von Map nutzen. Dadurch wird aber auch der Typ MapIterator sowie seine Implementierung für jede Implementierung von Map wiederverwendbar.

private find(K key) : Die private Methode MapImpl.find(K) sucht den entsprechenden Wert zu einem Schlüssel.

```
1  ...
2  private Number find(K key) {
3    Number hashCode = TJObjects.hashCode(key);
4    Number index = positiveMod(hashCode, modulo);
5    Number i = 1;
6    Boolean found = false;
7    while (i < modulo) {
8      if (index < entries.size()) {
9        MapEntry<K, V> candidate = entries[index];
10       if (candidate != null && candidate.getKey() == key) {
11         return index;
12       }
13     }
14     index = positiveMod(hashCode + i, modulo);
15     i += 1;
16   }
17   return null;
18 }
19 ...
```

Quellcode 4.17: MapImpl.tj

Die Nutzung von List, zur Implementierung einer Hashtabelle wird einem erfahrenen Entwickler erst einmal fremd vorkommen, da List keine feste Größe hat. TeaJay hat jedoch bewusst auf einen eingebauten Arraytyp (mit fester Größe) verzichtet. Durch Verwalten der Tabellengröße in einem eigenen Feld ist es auch kein all zu großer Aufwand, List für diesen Fall wie ein Array zu nutzen.

Zugriffsoperator : Zusätzlich zu den Methoden put(K, V) und get(K) bietet Map den Zugriffsoperator für den Zugriff auf die Tabelle an. Damit ist es möglich Folgendes zu schreiben:

```
1  Map<String, String> strMap = new Map<String, String>();
2  strMap["a"] = "b";
3  IO.println(strMap["a"]);
```

Ein Testtreiber für Map befindet sich unter den Lehrbeispielen auf der CD-ROM (map.MapTest).

4.1.2.3. Der Typ NumberStream

Bei diesem Typ geht es vor allem um die Diskussion, wann es sinnvoll ist, Operatorüberladung einzusetzen.

Stream :

```
1  package streams;
2  public typedef Stream<T> {
3    Stream(Closure<T>() spec);
4    T next();
5    T prev();
6  }
```

Quellcode 4.18: Stream.tj

NumberStream :

```
1  package streams;
2  public typedef NumberStream extends Stream<Number> {
3    NumberStream(Closure<Number>() spec);
4
5    static Stream<Number> operator_add(NumberStream stream1, Stream<Number> stream2);
6    static Stream<Number> operator_add(Stream<Number> stream1, NumberStream stream2);
7  }
```

Quellcode 4.19: NumberStream.tj

Auf den ersten Blick scheint das Überladen von + eine gute Idee zu sein: Für zwei Ströme a und b soll a + b einen neuen Stream erzeugen, welcher die Berechnungsvorschrift a.next() + b.next() hat. Dieses Verhalten wird nun für NumberStream implementiert:

```
1  package implementations;
2  import streams.*;
3  typeimpl NumberStreamImpl implements NumberStream {
4    NumberStreamImpl(Closure<Number>() spec) {
5      super(spec);
6    }
7    static Stream<Number> operator_add(NumberStream stream1, Stream<Number> stream2) {
8      return new NumberStream(closure[Stream<Number> stream1 = stream1;
9          Stream<Number> stream2 = stream2;]<Number>() {
10       return stream1.next() + stream2.next();
11     });
12   }
13   static Stream<Number> operator_add(Stream<Number> stream1, NumberStream stream2) {
14     //Analog
15   }
16 }
```

Quellcode 4.20: NumberStreamImpl.tj

Es ist nun möglich zwei Ströme zu addieren:

```
1    NumberStream a = new NumberStream(...);
2    NumberStream b = new NumberStream(...);
3    Stream<Number> a_plus_b = a + b;
4    IO.println(a_plus_b.next()); //inkrementiert a und b
5    a.next(); //inkrementiert nur a
```

Der hier definierte Additionsoperator scheint nebenwirkungsfrei. Dies ist jedoch nicht ganz richtig: Wird auf a_plus_b die Methode next() aufgerufen, so werden auch die Ströme a und b inkrementiert. Umgekehrt wird, wenn z.B. a.next() aufgerufen wird, nur a inkrementiert, was beim nächsten Aufruf von a_plus_b.next() sichtbar wird. Dieses Beispiel soll zeigen, dass man bei der Nutzung von Operatorüberladung besonders vorsichtig sein sollte. Die oben beschriebenen Eigenschaften der Addition von Streams sind natürlich Artefakte der Implementierung von Stream. Es wäre durchaus denkbar, eine Addition zu implementieren, welche die Zustände ihrer unterliegenden Ströme nicht verändert. In solchen Fällen ist es besonders wichtig, das Verhalten der

Operationen präzise in der Typdefinition anzugeben (z.B. ob + Nebenwirkungen hat und wenn ja welche).

4.1.3. TeaJays funktionale Elemente

TeaJay ist zwar hauptsächlich eine imperative Sprache, hat jedoch, wie viele moderne Sprachen, auch funktionale Elemente. Im Folgenden wird vor allem auf Closures eingegangen: Closures tragen ihre Schnittstellenbeschreibung in ihrem Typnamen. Damit wird die Schnittstellenbeschreibung identitätsstiftend für einen Closure-Typ. Sie sind vergleichbar mit anonymen Implementierungen von Interfaces in Java, mit der Einschränkung, dass nur eine Methode definiert werden darf. Zudem wird der Typ eines Closures nur über seine Signatur definiert und nicht zusätzlich über einen Namen. Closures sollten immer dann zum Einsatz kommen, wenn der Programmierer die Kontrolle über eine Implementierung nicht an das Laufzeitsystem abgeben will, weil er sehr spezielles Verhalten implementiert, z.B. bei Callbacks. Dieser Abschnitt möchte anhand von Beispielen zeigen, wie man versuchen kann, das Interesse der Lernenden am funktionalen Programmieren zu erwecken.

4.1.3.1. Closures als Methodenzeiger

Closures können in TeaJay als Methodenzeiger dienen:

```
1  Closure<Number>(String) strToNum = closure<Number>(String str) {
2    return Number.parseNumber(str);
3  };
```

strToNum kann nun wie ein Zeiger auf die Methode Number.parseNumber(String) verwendet werden und an jeder Stelle eingesetzt werden, an der ein Closure<Number>-(String) erwartet wird. Eine Einschränkung ist, dass Closures nicht auf private Methoden ihrer Umgebung zugreifen können und es somit nicht möglich ist, mit ihnen Zeiger auf private Methoden zu realisieren.

Möchte man Zeiger auf öffentliche Instanzmethoden realisieren, so kann die entsprechende Instanz in die Umgebung des Closures aufgenommen werden:

```
1  List<String> lst = ...;
2  Closure(String) addToLst = closure[List<String> lst = lst;](String str) {
3    lst.add(str);
4  }; //Zeigt nun auf die konkrete Instanzmethode lst.add(String)
```

Eine andere Möglichkeit wäre das Instanzobjekt als Argument des Closures zu erwarten:

```
1  List<String> lst = ...;
2  Closure(String) addToLst = closure[](List<String> lst, String str) {
3    lst.add(str);
4  }; //Zeigt auf die allgemeine Instanzmethode List<String>.add(String)
```

Damit lassen sich Methoden beinahe wie *First-Class-Objekte* behandeln.

4.1.3.2. Der Typ MapReducer

Mithilfe von Closures lässt sich ein prominentes Beispiel funktionaler Programmierung elegant in TeaJay formulieren:

Typ :

```
1  package mapreduce;
2  /*
3   * Definiert den Typen MapReducer, welcher generische Operationen für map bzw. reduce ↩
       ↪auf Listen bereitstellt.
4   */
5  public typedef MapReducer {
6    /*
7     * Reduziert list mit der Funktion func, wobei v0 der Wert der Reduktion der leeren ↩
         ↪Liste ist.
8     */
9    static <T,R> R reduce(Closure<R>(T, R) func, List<T> list, R v0);
10   /*
11    * Wendet die Funktion func auf jedes Element von list an und gibt die neu entstandene↩
         ↪ Liste zurück.
12    */
13   static <T,R> List<R> map(Closure<R>(T) func, List<T> list);
14 }
```

Quellcode 4.21: MapReducer.tj

Implementierung :

```
1  package implementations;
2
3  import mapreduce.MapReducer;
4
5  typeimpl MapReducerImpl implements MapReducer {
6
7    static <T,R> R reduce(Closure<R>(T, R) func, List<T> list, R v0) {
8      if (list.isEmpty()) {
9          return v0;
10     }
11     return func(/*head*/ list[0],
12       MapReducerImpl.<T,R>reduce(func, /*tail*/ list.subList(1), v0));
13   }
14
15   static <T,R> List<R> map(Closure<R>(T) func, List<T> list) {
16     List<R> copy = new List<R>();
17     for (T elem : list) {
18         copy.add(func(elem));
19     }
20     return copy;
21   }
22 }
```

Quellcode 4.22: MapReducerImpl.tj

Der Typ MapReducer bietet Funktionalität, um die Operationen map und reduce (im Sinne von Funktionen höherer Ordnung) auf Listen anzuwenden. Möchte man z.B. die Summe aller Elemente einer Liste von Zahlen berechnen, kann MapReducer in folgender Weise genutzt werden:

```
1  List<Number> aList = ...;
2  Number sum = MapReducer.<Number,Number>reduce(closure<Number>(Number a, Number b) {
3    //Zeiger auf die Methode Number.operator_add(Number, Number)
4    return a + b;
5  }, aList, 0);
```

Als ein weiteres Beispiel kann das Finden des Maximums einer Liste von Zahlen angeführt werden:

```
1  List<Number> aList = ...;
2  Number max = MapReducer.<Number,Number>reduce(closure<Number>(Number a, Number b) {
3    if (b == null || a > b) {
4      return a;
5    }
6    return b;
7  }, aList, null);
```

Oben gezeigte Beispiele zu entwickeln wäre auch als Übungsaufgabe für Lernende geeignet. Anhand von MapReducer können auch generische Methoden eingeführt werden.

4.1.3.3. Der Typ Selector

Eine Aufgabenstellung für einen Typ Selector könnte wie folgt lauten:

a) Entwerfen Sie einen Typ Selector, welcher die beiden beschriebenen generischen Methoden besitzt:

select : Wählt alle Elemente, die ein Prädikat erfüllen, aus einer Liste aus und gibt eine Liste all jener Elemente zurück.

retain : Löscht alle Elemente aus einer Liste welche ein Prädikat nicht erfüllen.

b) Implementieren Sie o.g. Typ ohne rekursive Aufrufe.

c) Implementieren Sie o.g. Typ durch Nutzen von Rekursion (ohne Verwendung von Schleifenanweisungen).

d) Implementieren Sie o.g. Typ mithilfe von mapreduce.MapReducer.

Typ :

```
1  package select;
2
3  public typedef Selector {
4    /*
5     * Erzeugt eine neue Liste und fügt alle Elemente die das Prädikat erfüllen hinzu.
6     */
7    static <T> List<T> select(Closure<Boolean>(T) predicate, List<T> in);
8
9    /*
10    * Löscht alle Elemente aus inAndOut, welche das Prädikat nicht erfüllen.
11    */
12   static <T> void retain(Closure<Boolean>(T) predicate, List<T> inAndOut);
13 }
```

Quellcode 4.23: Selector.tj

Beispielimplementierung b) stellt den wohl direktesten Weg zum Ziel dar.

```
1  package implementations;
2  import select.Selector;
3  typeimpl SelectorImpl implements Selector {
4    static <T> List<T> select(Closure<Boolean>(T) predicate, List<T> in) {
5      List<T> out = new List<T>();
6      for (T elem : in) {
7        if (predicate(elem)) {
8          out.add(elem);
9        }
10     }
11     return out;
12   }
13   static <T> void retain(Closure<Boolean>(T) predicate, List<T> inAndOut) {
14     TJIterator<T> it = inAndOut.iterator();
15     while (it.hasNext()) {
16       if (!predicate(it.next())) {
17         it.remove();
18       }
19     }
20   }
21 }
```

Quellcode 4.24: SelectorImpl.tj

Beispielimplementierung c) : Die auf Nebenwirkungen angewiesene Methode retain lässt sich rekursiv nicht allzu elegant formulieren. Für select ist das dagegen möglich (das heißt nicht, dass der Code performanter ist):

```
1  package implementations;
2  import select.Selector;
3  typeimpl SelectorImpl2 implements Selector {
4    static <T> List<T> select(Closure<Boolean>(T) predicate, List<T> in) {
5      if (!in.isEmpty()) {
6        if (predicate(in[0])) {
7          return new List<T>(/*head*/ in[0], SelectorImpl2.<T>select(predicate, /*tail*/ ↩
               ↪in.subList(1)));
8        } else {
9          return SelectorImpl2.<T>select(predicate, in.subList(1));
10       }
11     }
12     return new List<T>();
13   }
14   static <T> void retain(Closure<Boolean>(T) predicate, List<T> inAndOut) {
15     SelectorImpl2.<T>retain(predicate, inAndOut, 0);
16   }
17   private static <T> void retain(Closure<Boolean>(T) predicate, List<T> inAndOut, Number↩
         ↪ index) {
18     if (index < inAndOut.size()) {
19       if (!predicate(inAndOut[index])) {
20         inAndOut.remove(index);
21         SelectorImpl2.<T>retain(predicate, inAndOut, index);
22       } else {
23         SelectorImpl2.<T>retain(predicate, inAndOut, index + 1);
24       }
25     }
26   }
27 }
```

Quellcode 4.25: SelectorImpl2.tj

Beispielimplementierung d) : Bei dieser Aufgabe muss sich ein Lernender näher mit Closures und dem generischen Typsystem befassen.

```
1  package implementations;
2  import select.Selector;
3  import mapreduce.MapReducer;
4  typeimpl SelectorImpl3 implements Selector {
5    static <T> List<T> select(Closure<Boolean>(T) predicate, List<T> in) {
6      Closure<List<T>>(T, List<T>) reduce = closure[Closure<Boolean>(T) predicate = ↩
           ↪predicate;]<List<T>>(T elem, List<T> lst) {
7        if (predicate(elem)) {
8          return new List<T>(/*head*/ elem, /*tail*/ lst);
9        } else {
10         return lst;
11       }
12     };
13     return MapReducer.<T, List<T>>reduce(reduce, in, /*leere Liste*/ new List<T>());
14   }
15   static <T> void retain(Closure<Boolean>(T) predicate, List<T> inAndOut) {
16     List<T> tmp = SelectorImpl3.<T>select(predicate, inAndOut);
17     inAndOut.clear();
18     inAndOut.addAll(tmp);
19   }
20 }
```

Quellcode 4.26: SelectorImpl3.tj

Funktionale Programme lassen sich in TeaJay nicht so elegant, wie in vorwiegend funktionalen Sprachen (z.B. Lisp oder Haskell) formulieren. Jedoch kann TeaJay durchaus dazu eingesetzt werden, Grundlagen der funktionalen Programmierung zu lehren.

4.1.4. TeaJays Typsystem

TeaJays statische Typisierung ist in einigen Aspekten stärker als die von Java. Jedoch gibt es Stellen, an denen sie schwächer erscheint: Zum Beispiel ist es nicht ohne weiteres möglich, statisch auszudrücken, dass eine Zahl ganz sein soll. Das liegt an

der Tatsache, dass Number der einzige eingebaute Zahltyp von TeaJay ist und für eine beliebige rationale Zahl stehen kann. TeaJays statische Typsystem ist aber formal stärker als das von Java, denn es gibt in TeaJay z.B. keine *raw types*[4]. Das starke und generische Typsystem soll den Lernenden dazu anregen, stärker über Typen nachzudenken. Das Weglassen von *raw types* wird vermutlich dazu führen, dass sich ein Lernender früher oder später mit den Wildcard-Typen auseinandersetzen muss. In Java ist es nämlich, durch die Verwendung von *raw types* möglich, das generische Typsystem zu umgehen. Dabei wird jedoch immer *heap pollution* [GJS+12, Paragraph 4.8] riskiert, da eine statische Typprüfung umgangen wird. TeaJays Typsystem hat zudem wenige Ausnahmen, vor allem da Arrays und primitive Typen weggefallen sind. Dies soll TeaJay als Lehrsprache zu Gute kommen.

Der Typ Number soll vor allem für Anfänger den Umgang mit Zahlen intuitiver machen. In TeaJay besteht z.B. zwischen folgenden Ausdrücken kein Unterschied:

1 / 2 1.0 / 2.0 1 / 2.0

Ihr Ergebnis ist vom Typ Number und wird als $\frac{1}{2}$ dargestellt. Möchte ein Entwickler ganzzahlige Division erzwingen, so muss er z.B. das Ergebnis von 1 / 2 explizit in eine ganze Zahl umwandeln:

```
1 (1 / 2).toInteger() //== 0
```

toInteger() schneidet dabei, anders als floor(), den Fließkommaanteil einfach ab und simuliert damit das bekannte Verhalten der Umwandlung von Fließkommazahlen zu ganzen Zahlen aus Java. Dabei verhält sich toInteger() für negative Zahlen wie ceil() und für positive Zahlen wie floor(). toInteger() soll die Portierung von Algorithmen, welche z.B. ganzzahlige Division nutzen, erleichtern.

4.1.4.1. List vs. Array

TeaJay besitzt, anders als viele andere imperative Sprachen, keinen Arraytyp, was für eine höhere Abstraktion der Hardware-Ebene sorgt. Die Unterschiede von TeaJays List-Typ zu einem klassischen Java-Array sind dabei nur gering, da List auch Zugriffe über einen 0-basierten Index erlaubt. Der hauptsächliche Unterschied besteht darin, dass List, anders als ein Array, keine festgelegte Größe hat sondern nach Bedarf wächst. Zum Beispiel ist Folgendes mit List möglich:

```
1 List<String> lst = new List<String>();
2 lst[8] = "acht";
3 IO.println(lst.size()); //9
4 IO.println(lst[6]); //null
```

Dabei wird der Inhalt der Plätze 0 bis 7 auf null gesetzt. Ein lesender Zugriff auf einen vorher uninitialisierten Bereich ist auch bei List verboten:

```
1 List<String> lst = new List<String>();
2 IO.println(lst[0]); //Wirft eine IndexOutOfBoundsException
```

Alternativ hätte das Ergebnis von lst[0] auch als null gewählt werden können. Jedoch ist es durchaus legitim null als Element einer Liste zu haben. Dadurch wäre die Unterscheidung zwischen einem Zugriff auf uninitialisierte Bereiche von einem Zugriff auf initialisierte Bereiche schwieriger. Zudem ist der sogenannte *Off-by-one*[5]-Fehler

[4] vgl. Abschnitt 3.4.3.4

[5] [Off13]

ein recht häufiger Fehler, welcher durch eine IndexOutOfBoundsException schneller als solcher erkannt werden kann.

Durch den Typ List wird auch eine Schwäche von Javas Arraykonzept umgangen: Der Umgang mit Java-Arrays kann, auch ohne einen expliziten Cast, zu *heap pollution* führen.

```
1  //Java
2  String[] strArr = new String[] {"a", "b"};
3  Object[] objArr = strArr; //Der Java-Compiler lässt diese Zuweisung zu.
4  objArr[0] = new Object(); //Typfehler erst zur Laufzeitfehler. objArr ist in Wirklichkeit ein↩
       ↪ String[].
5  //TeaJay
6  List<String> strLst = new List<String>("a", "b");
7  List<Object> objLst = strLst; //Kompilierzeitfehler! List<String> ist nicht ↩
       ↪zuweisungskompatibel zu List<Object>
8  //Wildcards einsetzen:
9  List<?> objLst = strLst; //OK
10 Object obj = objLst.get(0); //OK
11 objLst.set(0, new Object()); //Kompilierzeitfehler
```

Dieses Problem lässt sich aber, z.B. durch konsequentes Nutzen von java.util.ArrayList<T>, auch in Java umgehen.

TeaJays Listentyp lässt sich also ähnlich wie ein Array benutzen, jedoch auch wie eine Lisp-Liste. Typische t.Lisp-Operationen übersetzen sich wie folgt auf List:

t.Lisp	TeaJay
(cons head tail)	**new** List<...>(head, tail)
(car lst)	lst [0]
(cdr lst)	lst . subList (1)
(list a b ...)	**new** List<...>(a, b, ...)
(join lst1 lst2)	**new** List<...>(lst1, lst2)
(nil? lst)	lst .isEmpty()
nil	**new** List<...>()

Tabelle 4.1.: t.Lisp-Liste vs. TeaJay-Liste

Zusätzlich muss in TeaJay noch die Korrektheit der statischen Typisierung beachtet werden:

```
1  List<String> strLst = new List<String>("a", "b");
2  List<Object> objLst = new List<Object>(5, "w");
3  List<Object> join1 = new List<Object>(strLst, objLst); // OK
4  strLst.add("c"); // nicht sichtbar in join1
5  List<String> join2 = new List<String>(strLst, objLst); //Fehler
```

Diese Fähigkeiten von List vereinfachen zustandsloses Programmieren in TeaJay. Dies kann man auch in Abschnitt 4.1.3.3 sehen. Dennoch ist TeaJay nur bedingt für zustandsloses Programmieren geeignet, da es nach wie vor eine vorwiegend imperative Sprache ist und der Zustand von List jederzeit geändert werden kann. Daher braucht es immer noch viel Disziplin auf Seiten des Programmierers um zustandslos zu Programmieren.

4.2. TeaJay in der Praxis

TeaJay wurde mit dem sekundären Ziel entworfen, einen Mehrwert bei der Lösung praktischer Probleme zu bieten. Dieser Mehrwert soll darin bestehen, mehr Struktur in die Entwicklung von Software zu bringen und die *Design by contract* Methode zu fördern. Die Möglichkeit der Verbindung von Java- und TeaJay-Code ermöglicht es,

aus TeaJay auf eine Vielzahl von Bibliotheken zuzugreifen. In diesem Abschnitt werden auch einige Beispiele aufgeführt und diskutiert, die zeigen, wie man vorhandene Java-Bibliotheken in TeaJay nutzen kann. Die Möglichkeit eines TeaJay-Entwicklers auf, vorhandene Java-Bibliotheken zuzugreifen und die daraus resultierende Mächtigkeit der Sprache, haben auch Auswirkungen auf TeaJay als Lehrsprache: Ein fortgeschrittener Lernender wird nicht durch Grenzen der Sprache demotiviert, selbst komplexere Software zu entwickeln. Sichtbare Erfolge (z.B. eine grafische Oberfläche) bieten oft ein größeres Erfolgserlebnis als einfache Konsolenprogramme und könnten helfen, die Motivation des Lernenden aufrecht zu halten.

4.2.1. ADTs in TeaJay

In 4.1.1 konnte man bereits erkennen, dass TeaJay die Spezifikation eines Typs von seiner Implementierung trennt. Das bedeutet für den Klienten, dass er beim Instanzieren eines Typs nur den Namen seiner Spezifikation kennen muss. TeaJay verbietet es sogar direkt auf eine Implementierung zuzugreifen (mit wenigen Ausnahmen). Ein Klient kann also nur auf Operationen zugreifen, die in der ADT-Spezifikation vorkommen (insbesondere kann er nicht auf Felder zugreifen, da diese von TeaJay nicht als Teil der öffentlichen Schnittstelle gesehen werden).

Eine Implementierung ohne Spezifikation lässt sich nicht kompilieren. Dadurch wird der Entwickler dazu angeregt, sich zuerst mit der Spezifikation des Typs zu befassen, bevor er sich Gedanken um eine Implementierung macht. Nachdem eine Spezifikation existiert, kann die Aufgabe der Implementierung auch an einen anderen Entwickler weitergegeben werden, während ein Klientenprogramm für den neuen Typ bereits weiter formuliert werden kann. Unter Zuhilfenahme einer Dummyimplementierung könnte das Programm sogar bereits getestet werden. TeaJay unterstützt also Entwickler bei der modularen Entwicklung von Software, indem es dieses Vorgehen auf natürliche Weise in der Sprache unterstützt und nicht modulare Ansätze erschwert. Da TeaJay aber primär eine Lehrsprache ist, ist es möglich, dass erfahrene Entwickler die „Bevormundung“ durch TeaJay ablehnen.

4.2.2. Entwicklung einer einfachen Ein-/Ausgabe Bibliothek für TeaJay

Der TeaJay-Standardbibliothek fehlt momentan eine API für allgemeine Ein- und Ausgabeoperationen. Unter Zuhilfenahme der Java-Kompatibilitätsschicht wird in diesem Abschnitt eine solche entworfen.

teajay.io.TJOutputStream :

```
1   ...
2   /* Creates a new TJOutputStream that writes to file. */
3   TJOutputStream(File file) throws IOException;
4   ...
5   /* Writes the lowset 8 bits of the Two's complement representation of val. */
6   void writeByte(Number val) throws IOException;
7   /* Writes the lowset 16 bits of the Two's complement representation of val. */
8   void writeShort(Number val) throws IOException;
9   /* Writes the lowset 32 bits of the Two's complement representation of val. */
10  void writeInt(Number val) throws IOException;
11  ...
12  /* Closes the stream. */
13  void close() throws IOException;
14  ...
```

Quellcode 4.27: TJOuptutStream.tj

teajay.io.TJInputStream :

```
1  ...
2  /* Creates a new TJInputStream that reads from file. */
3  TJInputStream(File file) throws IOException;
4  ...
5  /* Reads four octets from the underlying stream and interprets it as an unsigned short↩
       ↪ */
6  Number readUnsignedInt() throws IOException;
7  /* Reads four octets from the underlying stream and interprets it as a signed int */
8  Number readSignedInt() throws IOException;
9  ...
10 /* Closes the stream */
11 void close() throws IOException;
12 ...
```

Quellcode 4.28: TJInputStream.tj

Die entworfenen Streams bieten Methoden, um 8-, 16-, 32- und 64-Bit vorzeichenlose und vorzeichenbehaftete ganze Zahlen zu verarbeiten und zwei Methoden, um jeweils Fließkommazahlen in einfacher und doppelter Genauigkeit zu schreiben bzw. zu lesen. Weiterhin erlauben die Streams die Verarbeitung von UTF-Strings und das Schreiben und Lesen des Typs Number. Dabei wird Number in einen String umgewandelt und mit writeUTF(String) geschrieben. Dieses Format ist nicht ideal, da writeUTF(String) maximal Strings der Länge 65535 schreiben kann. Bei weiterer Entwicklung der API sollte dieses gegen ein mächtigeres ersetzt werden. Die Implementierung nutzt die aus java.io bekannten Streams DataOutputStream bzw. DataInputStream. Es gilt zu beachten, dass die Methoden writeByte(Number), writeShort(Number), writeInt(Number), writeLong(Number) ganze Zahlen als Parameter erwarten und einen Laufzeitfehler auslösen, sollte dies nicht der Fall sein.

Im Folgenden werden interessante Details der Implementierung von TJOutputStream und TJInputStream gezeigt:

teajay.io.impl.TJOutputStreamImpl : Um das Bitmuster der acht niederwertigsten Bits zu schreiben, wird folgender Code verwendet (out ist vom Typ java.io.DataOutputStream):

```
1  void writeByte(Number n) throws IOException {
2    out.writeByte(JavaPrimitives.toInt(n.and(0xFF)));
3  }
```

Schwieriger wird es, das Bitmuster einer 32-Bit Ganzzahl zu schreiben, da der primitive Java-Typ int nur vorzeichenbehaftet zur Verfügung steht:

```
1  void writeShort(Number n) throws IOException {
2    out.writeShort(JavaPrimitives.toInt(n.and(0xFFFF)));
3  }
4  void writeInt(Number val) throws IOException {
5    this.writeShort(val.and(0xFFFF0000).shiftRight(16));
6    this.writeShort(val.and(0x0000FFFF));
7  }
```

Ähnlich wird in writeLong(Number) verfahren.

teajay.io.impl.TJInputStreamImpl : Die Implementierung ist vergleichsweise simpel. Als Beispiel dient das Lesen einer vorzeichenlosen 32-Bit Ganzzahl:

```
1  Number readUnsignedInt() throws IOException {
2    return this.readSignedInt().and(0xFFFFFFFF); //signed -> unsigned
3  }
4
5  Number readSignedInt() throws IOException {
6    return JavaPrimitives.toNumber(in.readInt());
7  }
```

Auf der CD-ROM befinden sich die vollständigen Implementierungen und zwei weitere Streams (teajay.io.TJWriter und teajay.io.TJReader), welche den Umgang mit Textdateien ermöglichen.

4.2.3. Implementierungen für TeaJays Standardbibliothek schreiben

Folgende Implementierungen von Typen der Standardbibliothek sind austauschbar:

- teajay.lang.List
- teajay.util.Math
- teajay.util.RandomGenerator
- teajay.util.Sorter

Im Laufe der Entwicklung von TeaJay kann sich diese Liste noch erweitern lassen. Als nächstes wird am Beispiel von teajay.util.Sorter gezeigt, wie eine Implementierung der Standardbibliothek ersetzt werden kann:

teajay.util.Sorter :

```
1  public typedef Sorter {
2    static <T> void sort(RandomAccess<T> list, Closure<Number>(T, T) comparator);
3    static <T extends TJComparable<? super T>> void sort(RandomAccess<T> list);
4  }
```

mysort.SorterImpl : Implementierung von QuickSort:

```
1  package mysort;
2  typeimpl SorterImpl implements teajay.util.Sorter {
3    public static <T extends TJComparable<? super T>> void sort(RandomAccess<T> list) {
4      SorterImpl.<T>sort(list, closure<Number>(T a, T b) {
5        return a.tjCompareTo(b);
6      });
7    }
8    public static <T> void sort(RandomAccess<T> list, Closure<Number>(T,T) comparator) {
9      SorterImpl.<T>qsort(list, comparator, 0, list.size() - 1);
10   }
11   private static <R> void swap(RandomAccess<R> list, Number i, Number j) {
12     R tmp = list.get(i);
13     list.set(i, list.get(j));
14     list.set(j, tmp);
15   }
16   private static <T> void qsort(RandomAccess<T> list, Closure<Number>(T,T) comparator, ↩
         ↪Number begin, Number end) {
17     ... //Quicksort
18   }
19 }
```

Quellcode 4.29: SorterImpl.tj

Um die neue Implementierung nutzen zu können, muss diese der Laufzeitumgebung als Ersatz bekanntgegeben werden. Dies kann z.B. durch eine Konfigurationsdatei (mysort/SoterImpl.cfg) geschehen:

```
teajay.util.Sorter = mysort.SorterImpl
```

Quellcode 4.30: SorterImpl.cfg

Diese Datei muss der Laufzeitumgebung mitgeteilt werden und mysort.SorterImpl muss sich im Suchpfad eben jener befinden. Einen Testtreiber namens mysort.SorterTest kann man mit der neuen Implementierung wie folgt starten:

```
> tj -cfg mysort/SorterImpl.cfg mysort.SorterTest test
```

Quicksort auf diese Weise zu implementieren kann auch als fortgeschrittene Übungsaufgabe dienen.

4.2.4. GUI-Programme mit TeaJay

TeaJay ermöglicht es, mit einigen Einschränkungen, das Swing-Framework zu nutzen, um Programme mit grafischer Benutzeroberfläche zu erstellen. In diesem Abschnitt wird am Beispiel eines Taschenrechners gezeigt, wie sich einfache grafische Benutzeroberflächen erstellen lassen.

ClosureActionListener :

```
package calc;

import java.awt.event.ActionListener;
import java.awt.event.ActionEvent;

public typedef ClosureActionListener interfaces ActionListener {
  ClosureActionListener(Closure(ActionEvent) action);
}
```

Quellcode 4.31: ClosureActionListener.tj

```
package calc.impl;

import java.awt.event.ActionEvent;
import calc.ClosureActionListener;

typeimpl ClosureActionListenerImpl implements ClosureActionListener {
  Closure(ActionEvent) action;

  ClosureActionListenerImpl(Closure(ActionEvent) act) {
    action = act;
  }

  public void actionPerformed(ActionEvent e) {
    action(e);
  }
}
```

Quellcode 4.32: ClosureActionListenerImpl.tj

Der ClosureActionListener ermöglicht es auf das Swing-Ereignis actionPerformed mit einem TeaJay-Closure zu reagieren.

GUICalculatorImpl : Das Layout der Zahleingabe wird mit einem java.awt.GridBagLayout realisiert, wobei man aus TeaJay heraus auf Java-Felder zugreifen muss:

```
  void run() {
  [...]
  JFrame frame = new JFrame("TeaJay - Calc");
  frame.setLayout(new BorderLayout());
  frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE /*statischer Feldzugriff*/);
  JTextField out = new JTextField("0");
  [...]
  Display disp = new Display(out);
```

```
9   ClosureActionListener digitListener = new ClosureActionListener(
10    closure[Display display = disp;](ActionEvent evt) {
11      display.input(evt.getActionCommand());
12      return;
13    }
14  );
15  [...]
16  GridBagConstraints constr = new GridBagConstraints();
17  constr.ipadx = JavaPrimitives.toInt(25); //Feldzugriff
18  constr.ipady = JavaPrimitives.toInt(25);
19  while (i >= 0) {
20    JButton button = new JButton(i.toString());
21    button.addActionListener(digitListener);
22    button.setFont(buttonFont);
23    button.setActionCommand(button.getText());
24    Tuple<Number, Number> grid = gridMap(i);
25    constr.gridx = JavaPrimitives.toInt(grid.getFirst());
26    constr.gridy = JavaPrimitives.toInt(grid.getSecond());
27    if (i == 0) {
28      constr.gridwidth = JavaPrimitives.toInt(2);
29    }
30    constr.fill = GridBagConstraints.BOTH;
31    keys.add(button, constr);
32    i -= 1;
33  }
34  [...]
35  frame.pack();
36  frame.setVisible(JavaPrimitives.toBoolean(true)); //Zeige das Fenster
37  }
```

Quellcode 4.33: GUICalculatorImpl.tj

Das komplette Programm befindet sich im Anhang der Arbeit (auf CD-ROM), es kann mit dem Befehl

```
1   > tj calc.GUICalculator run
```

ausgeführt werden. An diesem Beispiel kann man gut erkennen, dass sich viele Java-Klassen und Methoden relativ natürlich in TeaJay nutzen lassen.

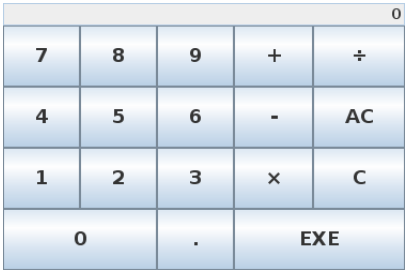

Abbildung 4.1.: GUICalculator

Einen Taschenrechner mit grafischer Benutzeroberfläche zu entwickeln wäre auch als Lehraufgabe für Fortgeschrittene denkbar. Um aber grafische Benutzeroberflächen natürlich in TeaJay entwickeln zu können, wird zuerst ein GUI-Framework für TeaJay benötigt.

4.2.5. Numerische Berechnungen mit Number

Beim Programmieren mit TeaJay ist aufgefallen, dass viele numerische Verfahren nicht dazu geeignet sind mit Number durchgeführt zu werden, da der für die Darstellung der Zahlen benötigte Speicher schnell anwächst. Es ist daher nötig in regelmäßigen Abständen die Genauigkeit der Darstellung herabzusetzen (z.B. durch Number.round(Number)). Das Heron-Verfahren zur Berechnung der Quadratwurzel könnte in TeaJay wie folgt aussehen:

```
1  package implementations;
2  /* Heron Verfahren */
3  typeimpl HeronImpl implements heron.Heron {
4    static Number sqrt(Number x, Number iterations) {
5      Number result = (x + 1) / 2;
6      Number i = 0;
7      while (i < iterations) {
8        result = (result + (x / result)) / 2;
9        i += 1;
10     }
11     return result;
12   }
13 }
```

Quellcode 4.34: HeronImpl.tj

Führt man viele Iterationen des Verfahrens durch, wächst die Größe der Darstellung von result extrem schnell an. Daher sollte result in regelmäßigen Abständen gerundet werden. Dabei muss ein Entwickler von numerischen Verfahren einerseits dafür sorgen, dass die Darstellungen von verwendeten Zahlvariablen nicht zu groß werden und andererseits sicherstellen, dass eine vorgegebene Fehlergrenze unterschritten wird. Dazu wird die Implementierung des Heron-Verfahrens um die Möglichkeit eine Genauigkeit anzugeben erweitert:

```
1  package implementations;
2  /* Heron Verfahren */
3  typeimpl HeronExtendedImpl implements heron.HeronExtended {
4    static Number sqrt(Number x, Number precision) {
5      Number threshold = (10).pow(-precision);
6      Number result = (x + 1) / 2;
7      Number last = null;
8      do {
9        last = result;
10       result = result.round(precision + 1);
11       result = (result + (x / result)) / 2;
12     } while ( (result - last).abs() > threshold );
13     return result;
14   }
15 }
```

Quellcode 4.35: HeronExtendedImpl.tj

Number.round(Number) wird aufgerufen, um die Darstellung der Zahlen klein zu halten. In anderen Programmiersprachen wird die Genauigkeit von den verwendeten Typen (z.B. double oder float) vorgegeben. In TeaJay sollte bei numerischen Berechnungen immer eine Genauigkeit angegeben werden.

Es gilt des Weiteren zu beachten, dass die arithmetischen Operationen von Number keine konstante Laufzeit haben, sondern abhängig von der Größe der Zahldarstellung sind[6].

Auf der CD-ROM befindet sich ein weiteres Beispiel für numerische Berechnungen in TeaJay: Ein Typ zur statischen Visualisierung der Mandelbrotmenge mandelbrot.MandelbrotImager mit einem Testtreiber (mandelbrot.MandelbrotTester) und zwei Implementierungen:

implementations.MandelbrotImagerImpl nutzt TeaJays Number-Typ zur Berechnung der Mandelbrotmenge.

implementations.MandelbrotImagerFastImpl nutzt teajay.util.Real zur Berechnung der Mandelbrotmenge.

Das Beispiel lässt sich wie folgt starten:

[6]Die Laufzeiten der Operationen von Number hängen direkt von den Laufzeiten der Operationen von java.math.BigInteger [jav13] ab.

```
> tj mandelbrot.MandelbrotTester run [<Breite> <Höhe> [<↩
    ↪Ausgabedatei>]]
```

Die Ausgabedatei wird im PNG-Format geschrieben. Mithilfe von -Ximpl kann man die beiden Implementierungen jeweils testen. Die Nutzung der schnellen Implementierung kann man wie folgt erzwingen:

```
> tj -Ximpl mandelbrot.MandelbrotImager=implementations.↩
    ↪MandelbrotImagerFastImpl mandelbrot.MandelbrotTester run ...
```

4.2.6. Zählschleife nachrüsten

TeaJay besitzt keine klassische for-Schleife, sondern nur eine for-each-Schleife. Durch einen Typ Range kann diese jedoch einfach als eine vielseitige Zählschleife verwendet werden:

```
package range;
/* Implementiert die bekannte range()-Funktion für TeaJay.
 * Damit wird folgender Code möglich: for (Number n : Range.range(0, 8)) { ... } */
public typedef Range {
  /* Benutzt die Schrittgröße 1. */
  static TJIterable<Number> range(Number begin, Number end);
  /* Sollte begin > end sein, so wird absteigend iteriert und step negiert.
   * step muss immer positiv und größer als 0 angegeben werden. */
  static TJIterable<Number> range(Number begin, Number end, Number step);
}
```

Quellcode 4.36: Range.tj

Durch Range lassen sich viele Anwendungsfälle der klassischen for-Schleife abdecken, z.B. das Umkehren einer Liste:

```
List<String> lst = new List<String>("a", "b", "c", "d");
List<String> reversedList = new List<String>();
for (Number i : Range.range(lst.size() - 1, 0)) {
  reversedList.add(lst[i]);
}
IO.println(reversedList);
```

5. Implementierung

MB
MK

Die Implementierung von TeaJay gliedert sich in drei Projekte:

- ein Bytecode-Framework, welches die JVM-Spezifikation (Version 7) umsetzt
- eine Laufzeitumgebung für TeaJay-Programme
- ein Compiler, welcher TeaJay nach JVM-Bytecode übersetzt

Eine umfassende Beschreibung der Projekte würde den Rahmen dieser Arbeit sprengen, sodass nur auf die wichtigsten Aspekte der Implementierung eingegangen wird.

5.1. ByteCode-Framework

MK

Es gibt bereits viele - mehr oder weniger frei verfügbare - Frameworks, die es erlauben JVM-Bytecode zu generieren. Viele der Frameworks haben zu Beginn der Entwicklung an TeaJay aber entweder noch nicht Java 7 unterstützt und damit auch nicht die Instruktion invokedynamic, welche essenziell bei der Implementierung von TeaJay ist, oder waren nicht frei verfügbar. Daher wurde im Rahmen dieses Projekts ein eigenes Framework entwickelt. Das ByteCode-Framework orientiert sich an der JVM 7 Spezifikation [LYBB12] und ist nicht dazu entworfen worden ältere Class-File Versionen korrekt zu schreiben, obwohl es in der Lage ist, diese zu lesen. Eine weitere Einschränkung ist, dass eingelesene Code-Attribute nicht decodiert werden und in der Folge nicht modifiziert werden können. Es ist möglich diese Funktionalität nachzurüsten, jedoch wird sie für die Übersetzung von TeaJay nicht benötigt. Im Folgenden werden auch nur Besonderheiten dieses Frameworks aufgezeigt, wobei für detaillierte Informationen zum Format eines Class-Files auf die JVM-Spezifikation (vgl. [LYBB12]) verwiesen wird.

Die JVM ist eine Stapelmaschine, die zur Verwaltung von lokalen Variablen eine spezielle Tabelle benutzt. Der zugrundeliegende Stack wird nur für die Verwaltung von Operanden eingesetzt. Instruktionen holen ihre Operanden vom Stack, direkt aus dem *code array* (siehe 5.1.3.1) oder aus dem Konstantenpool (siehe 5.1.2.2) und legen ein eventuell vorhandenes Ergebnis auf dem Stack ab. Einige Instruktionen können lokale Variablen direkt in der lokalen Variablentabelle modifizieren. Auf lokale Variablen kann mit speziellen Instruktionen (z.B. aload) über einen 0-basierten Index zugegriffen werden.

5.1.1. Aufbau eines Class-File

Ein Class-File kodiert genau eine Java-Klasse und enthält alle Informationen die eine Java-Klasse ausmachen (vgl. [LYBB12, Kapitel 4]):

- Name der Klasse
- Zugriffsmodifizierer
- Name der Superklasse
- implementierte interfaces
- Konstanten (im Konstantenpool)
- Felder der Klasse
- Methoden der Klasse
- beliebige Attribute

Class-Files können auch für die Laufzeit unwichtige Informationen, wie z.B. generische Methodensignaturen, den Namen der Quellcodedatei oder Tabellen, die Anweisungen auf Zeilennummern abbilden, bereithalten.

5.1.2. Übersicht über das ByteCode-Framework

Das ByteCode-Framework bietet eine Abstraktion des JVM-ByteCodes und übernimmt sämtliche Adressberechnungen (ob Sprungadressen oder Konstantenpooladressen). Außerdem stellt es eine, auf einer niedrigen Abstraktionsebene angesiedelte, Zwischensprache für die JVM dar. Die Namen von Entitätsklassen orientieren sich direkt an den in der JVM-Spezifikation beschriebenen Namen. An einigen Stellen werden Plausibilitätsprüfungen durchgeführt und Verstöße mit einer teajay.bytecode.exceptions.ByteCodeException angezeigt. Dabei orientiert sich das Framework an der JVM-Spezifikation (vgl. [LYBB12])

5.1.2.1. ClassFile

Die Klasse teajay.bytecode.ClassFile steht als Entität für ein Class-File und verwaltet alle relevanten Informationen. Um Bytecode zu erzeugen, muss also zuerst ein Class-File erzeugt werden:

```
1 /* Erstellt ein leeres Class-File. Der Klassenname ist Foo und das Paket fun.
2    Die Superklasse ist, wenn nicht anders angegeben, vorerst java/lang/Object */
3 ClassFile classFile = new ClassFile("fun/Foo", ClassAccessFlag.SUPER, ClassAccessFlag.PUBLIC);
4 ...
5 classFile.write(...); //schreibt classFile in eine Datei oder in einen DataOutputStream
6 ...
7 ClassFile tmp = ClassFile.read(...); //liest ein Class-File ein
```

Qualifizierte Bezeichner werden für die JVM in einem anderen Format als auf der Sprachebene angegeben: Anstelle von "." als Separator, verwendet die JVM "/". Der Zugriffsmodifizierer SUPER sollte bei Java-Klassen immer gesetzt werden und verändert das Verhalten von invokespecial (vgl. [LYBB12, S. 72]). Eine Klasse mit dem unqualifizierten Namen Name muss, um direkt von der JVM genutzt werden zu können, in eine Datei mit dem Namen Name.class geschrieben werden.

Eine Entität die in ein Class-File geschrieben werden kann, muss das interface ByteCodeWritable implementieren:

```
1  package teajay.bytecode;
2  ...
3  public interface ByteCodeWritable {
4      public void write(DataOutputStream out) throws IOException, ByteCodeException;
5  }
```

Quellcode 5.1: ByteCodeWritable.java

Entitäten, die zusätzlich über eine Adresse referenziert werden können, müssen zusätzlich ein ähnliches interface, namens IndexWritable, mit der Methode writeIndex(DataOutputStream) implementieren. Anders als write(...) schreibt writeIndex(...) nicht die Entität selbst, sondern die Adresse der Entität. Dies kann z.B. die Adresse einer Konstanten (im Konstantenpool) oder der Index einer Instruktion im *code array* (vgl. 5.1.3.1) sein.

5.1.2.2. Konstantenpool

Der Konstantenpool eines Class-File beinhaltet Konstanten, welche im Bytecode oder in anderen Teilen des Class-File benutzt werden können (z.B. Zeichenketten oder Zahlen). Ein Konstantenpooleintrag wird durch seine Adresse (Index in den Konstantenpool) referenziert. (vgl. [LYBB12, Paragraph 4.4])

Der Konstantenpool eines ClassFile wird durch teajay.bytecode.cpool.ConstantPool implementiert und jedes ClassFile besitzt einen solchen. Jede Entität die im Konstantenpool vorkommen darf, wird durch eine entsprechende Klasse im Paket teajay.bytecode.cpool abgebildet. Eine solche Entität darf immer nur in einem Konstantenpool eines einzigen Class-File vorkommen, denn sie hält ihre Konstantenpooladresse als Attribut vor. Soll eine Entität mit demselben Wert in einen weiteren Konstantenpool geschrieben werden, so kann eine Kopie erzeugt werden (CPoolEntry.clone()). Dabei werden, bis auf die Konstantenpooladresse, auch alle von der Entität referenzierten Entitäten kopiert.

Um das Eintragen einer Entität in den Konstantenpool muss sich der Klient nicht selber kümmern. Er muss nur sicherstellen, dass dieselbe Entität nicht in mehreren Konstantenpools verwendet wird. Erstellt der Klient z.B. ein teajay.bytecode.cpool.StringInfo

```
1  StringInfo strInfo = new StringInfo("ein string");
```

so wird automatisch auch ein teajay.bytecode.cpool.Utf8Info mit dem Wert "ein string" erzeugt. Wird strInfo nun einem Konstantenpool hinzugefügt, so wird jenes Utf8Info diesem ebenfalls hinzugefügt. Generell sollte es niemals nötig sein, explizit eine Entität dem Konstantenpool hinzuzufügen. ConstantPool sorgt auch dafür, dass gleiche Entitäten nicht doppelt im Konstantenpool vorhanden sind und übernimmt die Adressierung (auf welche der Klient keinen direkten Einfluss hat).

Einige Konstantenpooleinträge wie z.B. StringInfo, IntegerInfo oder ClassInfo können mit den Instruktionen ldc bzw. ldc_w[1] direkt auf den Operanden-Stack der JVM gelegt werden. Ein ClassInfo hat dabei z.B. den Laufzeittyp java.lang.Class.

Die beiden Konstantenpoolentitäten MethodDescriptor und FieldDescriptor werden beide durch ein Utf8Info beschrieben und existieren daher nur auf Framework-Ebene. Beim Einlesen wird automatisch, anhand des Formats, entschieden, ob ein Utf8Info

[1] ldc_w akzeptiert einen 16-bit Konstantenpooladresse während ldc nur eine 8-bit Adresse akzeptiert

einen MethodDescriptor bzw. FieldDescriptor darstellt und eine entsprechende Entität erzeugt. Beide sind Spezialisierungen der Klasse Utf8Info.

FieldDescriptor beschreibt einen beliebigen JVM-Laufzeittypen und wird als Zeichenkette kodiert. Dabei beschreibt folgende Zeichenkette beispielsweise den Referenztyp List:

```
Lteajay/lang/List;
```

L gibt bekannt, dass ein Referenztyp folgt. Primitive Typen werden durch einzelne Buchstaben codiert. Der Typ void wird z.B. durch V codiert. Weitere Informationen finden sich in [LYBB12, Paragraph 4.3.2].

MethodDescriptor beschreibt die Laufzeitsignatur einer Methode:

```
(Lteajay/lang/List;I)[Ljava/lang/Object;
```

Diese Zeichenkette codiert die Signatur einer Methode mit dem Rückgabewert java.lang.Object[] und zwei Parametern teajay.lang.List und int. Weitere Informationen finden sich in [LYBB12, Paragraph 4.3.3].

Beim Schreiben eines ClassFile werden zuerst alle Entitäten, die das ClassFile verwaltet, aufgefordert ihre Konstaten dem Konstantenpool hinzuzufügen. Jede Entität, die eventuell Konstanten in den Konstantenpool eintragen muss, implementiert dazu folgendes interface:

```
1 package teajay.bytecode.cpool;
2 ...
3 public interface ConstantPoolAddable {
4     public void add(ConstantPool cpool) throws ByteCodeException;
5 }
```

Quellcode 5.2: ConstantPoolAddable.java

5.1.2.3. Attribute

Den meisten Entitäten außerhalb des Konstantenpools lassen sich Attribute anhängen. Die JVM-Spezifikation listet einige Attribute auf, die für die JVM von Bedeutung sind und weitere, welche nur für einen Compiler von Bedeutung sind. Es ist möglich eigene Attribute zu definieren, welche dann von einer JVM ignoriert werden müssen, sofern diese nicht in der Lage ist sie zu interpretieren. Das Framework bietet Implementierungen aller in [LYBB12, Paragraph 4.7] aufgelisteten Attribute und ermöglicht es benutzerdefinierte Attribute zu verwenden. Um ein eigenes Attribut zu entwickeln muss von der Klasse teajay.bytecode.attributes.AttributeInfo geerbt und die abstrakte Methode getLength() implementiert werden. getLength() muss dabei die Anzahl an Bytes zurückgeben, die das Attribut verbrauchen wird. Zudem sollten die Methoden add(ConstantPool) und write(DataOutputStream) überschrieben werden, um benutzerdefinierte Daten schreiben zu können. Dabei ist zu beachten, dass immer zuerst die Supermethode aufgerufen werden sollte. Zudem muss eine Implementierung von AttributeInfo immer dessen Superkonstruktor aufrufen und dadurch einen Namen für das neue Attribut spezifizieren.

```
class MyAttribute extends AttributeInfo {
  public MyAttribute() {
    super(new Utf8Info("MyAttribute"));
  }
  ...
  @Override
  public void write(DataOutputStream out) throws IOException, ByteCodeException {
    super.write(out); //Writes the size of the attribute (getLength())
    //Nutzdaten schreiben
  }
  @Override
  public void add(ConstantPool cpool) throws ByteCodeException {
    super.add(cpool);
    //Hier die verwendeteten Konstanten eintragen
  }
}
```

Um benutzerdefinierte Attribute lesen zu können, müssen diese vorher via registerAttribute(Class<? extends AttributeInfo>) bei teajay.bytecode.attributes.AttributesReader registriert werden. Zudem muss das neue Attribut ein Feld mit dem Namen des Attributs folgendermaßen definieren:

```
  public static final String NAME = "MyAttribute";
```

Dabei muss der Name des Attributs nicht zwingend mit dem Namen der implementierenden Klasse übereinstimmen. Bei der Wahl des Namens sollte jedoch auf Namenskonflikte mit bereits vorhandenen Attributen geachtet werden. Des Weiteren muss eine der beiden folgenden Methoden vorhanden sein:

```
public static AttributeInfo read(DataInputStream in,ConstantPool cpool,
  int major, int minor,int length,Utf8Info name)
```

Dabei können hier aus in nicht mehr als length Bytes gelesen werden.

```
public static AttributeInfo read(byte[] data,ConstantPool cpool,
  int major,int minor,Utf8Info name)
```

Diese Methoden werden aufgerufen, um ein Attribut, welches einen Namen angibt der mit dem Wert von NAME übereinstimmt, zu lesen. Es folgt ein simples Beispiel anhand des Attributs SourceFile:

```
package teajay.bytecode.attributes;
import java.io.*;
import teajay.bytecode.U4;
import teajay.bytecode.cpool.*;
import teajay.bytecode.exceptions.ByteCodeException;
import static teajay.bytecode.util.Tools.ccpt; //Typprüfung für Konstantenpooleinträge
public final class SourceFile extends AttributeInfo {
    public static final String NAME = "SourceFile";
    private final Utf8Info sourceFile;
    private SourceFile(Utf8Info name, Utf8Info sourceFile) {
        super(name);
        checkName(NAME, name); //siehe AttributeInfo.checkName(...)
        this.sourceFile = sourceFile;
    }
    public SourceFile(String sourceFile) {
        this(new Utf8Info(NAME), new Utf8Info(sourceFile));
    }
    @Override
    public U4 getLength() throws ByteCodeException {
        return new U4(2);
    }
    public String getSourceFile() {
        return this.sourceFile.getValue();
    }
    @Override
    public void write(DataOutputStream out) throws ByteCodeException, IOException {
        super.write(out);
        sourceFile.writeIndex(out); //Adressen von Konstantenpooleinträgen sind immer zwei ↩
            ↪bytes lang
    }
    @Override
    public void add(ConstantPool cpool) throws ByteCodeException {
        super.add(cpool); //Fügt name hinzu
        sourceFile.add(cpool);
    }
```

```
35      public static SourceFile read(DataInputStream in, ConstantPool cpool, int major, int minor,↩
            ↪ int length, Utf8Info name) throws ByteCodeException, IOException {
36        //ccpt führt eine Typprüfung mithilfe der Reflection-API durch
37          Utf8Info sourceFile = ccpt(cpool.get(in.readUnsignedShort()), Utf8Info.class);
38          return new SourceFile(name, sourceFile);
39      }
40  }
```

Quellcode 5.3: SourceFile.java

Implementierungen aller Attribute, welche in [LYBB12, Paragraph 4.7] beschrieben sind, finden sich im Paket teajay.bytecode.attributes und dessen Unterpaketen.

5.1.2.4. Das Signature-Attribut

Das Signature-Attribut dient dazu, die Signatur eines Feldes, einer Klasse oder einer Methode zu überschreiben. Das Attribut dient der Umsetzung von Javas generischem Typsystem und wird nur vom Compiler ausgewertet, um eine Typprüfung durchführen zu können. Die Signatur wird dabei von einer speziell formatierten Zeichenkette angegeben. Dieses Attribut wird auch vom TeaJay-Compiler genutzt, um Typinformationen abzulegen. Für mehr Informationen zum Format einer Signatur siehe [LYBB12, Paragraph 4.3.4].

5.1.2.5. Felder und Methoden

Felder können einem ClassFile mithilfe von teajay.bytecode.fields.FieldInfo hinzugefügt werden:

```
1   //Erzeugt ein konstantes Feld mit dem Wert "string value"
2   FieldInfo constantField = new FieldInfo("constantField",
3     FieldDescriptor.fromClass(String.class), FieldAccessFlag.PRIVATE, FieldAccessFlag.FINAL);
4   classFile.addField(constantField);
5   constantField.addAttribute(new ConstantValue("string value"));
6   //Erzeugt ein privates Feld ohne Wert (null)
7   FieldInfo someField = new FieldInfo("someField",
8     FieldDescriptor.fromString("java/lang/Object"), FieldAccessFlag.PRIVATE);
9   classFile.addField(someField);
```

Methoden können einem ClassFile mithilfe der Klasse teajay.bytecode.methods.MethodInfo hinzugefügt werden. Das folgende Beispiel fügt classFile eine main-Methode hinzu.

```
1  MethodInfo main = new MethodInfo("main",
2    new MethodDescriptor(FieldDescriptor.getVoid(), FieldDescriptor.fromClass(String[].class)),
3      classFile.getThis(), MethodAccessFlag.PUBLIC, MethodAccessFlag.STATIC);
4  classFile.addMethod(main);
```

Solange nicht die Zugriffsmodifizierer NATIVE oder ABSTRACT gesetzt werden, bekommt ein MethodInfo bei der Erzeugung automatisch ein leeres CodeAttribute, auf welches man durch MethodInfo.getCode() zugreifen kann. Das CodeAttribute hat eine Sonderstellung und taucht nicht in der normalen Attributsliste von MethodInfo auf. Die Argumente der Methode stehen dem Bytecode in der lokalen Variablentabelle zur Verfügung.

Auf Class-File-Ebene werden Methoden durch ihre Parameterliste, den Namen und Rückgabetyp identifiziert. Das heißt in einem ClassFile dürfen mehrere Methoden mit derselben Parameterliste und demselben Namen vorkommen, wenn sich ihre Rückgabetypen unterscheiden. Dabei kann eine Methode auch nur von einer Methode mit derselben Laufzeitsignatur überschrieben werden. Folgendes Beispiel verdeutlicht dies:

```
1 class A {
2   public Object foo(int a) { ... }
3 }
4 class B extends A {
5   public String foo(int a) { ... }
6 }
```

Um zu erreichen dass B.foo(int) die Methode A.foo(int) überschreibt, muss eine sogenannte Bridge-Methode erzeugt werden, die wie folgt aussieht:

```
1 class B extends A {
2   public String foo(int a) { ... }
3   public /*bridge*/ Object foo(int a) { //Bridge-Methode
4     aload_0 // this referenz
5     iload_1 // int a
6     invokespecial B.foo(I)Ljava/lang/String; //Ruft die Methode foo mit dem Rückgabetyp String ↩
          ↪auf.
7     areturn // return the result
8   }
9 }
```

Die Bridge-Methode überschreibt A.foo(int) und hat den Zugriffsmodifizierer BRIDGE [LYBB12, S. 99] gesetzt. Dieser kann nicht durch Javas Sprachmittel gesetzt werden. Bridge-Methoden spielen auch eine Rolle bei der Implementierung von Javas generischem Typsystem, da Laufzeitsignatur und Kompilierzeitsignatur häufig nicht identisch sind aber die Methoden sich dennoch überschreiben sollen. Zum Beispiel muss in folgendem Fall ebenfalls eine Bridge-Methode erzeugt werden:

```
1 class A<T> {
2   public void add(T a) {} //Laufzeitsignatur: void add(Object a)
3 }
4 class B extends A<String> {
5   public void add(String a) {...} //soll add(T a) überschreiben
6   public /*bridge*/ void add(Object a) { //Bridge Methode
7     this.add((String) a);
8   }
9 }
```

Die Erzeugung von Bridge-Methoden kann dabei nicht vom Framework übernommen werden.

5.1.2.6. Annotationen

Das Bytecode-Framework erlaubt es mit Annotationen zu arbeiten. Dabei unterscheidet die JVM zwischen Annotationen die zur Laufzeit sichtbar sind und solchen die es nicht sind. Dazu gibt es vier Arten von speziellen Attributen, welche Annotationen vorhalten können (vgl. [LYBB12, Paragraph 4.7.16 - 4.7.20]):

Runtime[In]VisibleAnnotations : Diese Attribute dürfen in der Attributsliste eines Class-File, einer Methode oder eines Felds vorkommen. Je nachdem in welchem der beiden Attribute die Annotation vorkommt, muss sie zur Laufzeit sichtbar sein oder nicht. Die Sichtbarkeit wird aber bei der Definition der Annotation festgelegt.

Runtime[In]VisibleParameterAnnotations : Diese Attribute dürfen nur in der Attributsliste einer Methode vorkommen und dienen dazu die Parameter (und nicht die Methode selber) einer Methode zu annotieren.

Obigen Attributen können Instanzen der Klasse teajay.bytecode.attributes.annotations.-Annotation hinzugefügt werden:

```
MethodInfo main = ...;
RuntimeInvisibleAnnotations anns = new RuntimeInvisibleAnnotations();
main.addAttribute(anns);
Annotation ann = new Annotation(FieldDescriptor.fromString("teajay/util/TJSignature"));
anns.addAnnotation(ann);
ann.addElementValue("value", new ElementValue("([Ljava/lang/String;)V")); //Elementtyp: String
```

Die Klasse ElementValue bietet Konstruktoren für alle erlaubten Werte in einer Annotation. Dazu zählen u.a. alle primitiven Java-Typen, Enums und Strings. Die Methode main wurde mit teajay.util.TJSignature annotiert. Dieses wurde in Java wie folgt definiert:

```
package teajay.util;
public @interface TJSignature {
    String value();
}
```

Quellcode 5.4: TJSignature.java

Es wird bis auf folgende Ausnahmen wie ein normales interface übersetzt:

- Der Zugriffsmodifizierer ANNOTATION (ClassAccessFlag.ANNOTATION) wird gesetzt.
- Das interface erbt von java.lang.annotation.Annotation.
- Methoden können das AnnotationDefault-Attribut tragen um Standardwerte zu definieren.

5.1.3. Das Code-Attribut

Das Code-Attribut wird durch teajay.bytecode.attributes.code.CodeAttribute implementiert und ist wohl das umfangreichste Attribut des Frameworks. Durch das Code-Attribut wird der JVM, der eine Methode implementierende Code, bekanntgegeben. Jedes Code-Attribut erhält automatisch eine StackMapTable, welche jedoch nur geschrieben wird, falls sie nicht leer ist. CodeAttribute übernimmt auch die Verwaltung der *exception table* [LYBB12, S. 107ff] und erlaubt das Hinzufügen und Entfernen von Ausnahmebehandlern. Mit folgender Methode können z.B. Bereiche definiert werden, die von einem Ausnahmebehandler geschützt werden:

```
public void addExceptionHandler(JVMInstruction startPC, JVMInstruction endPC,
  JVMInstruction handlerPC, ClassInfo catchType) { ... }
```

startPC und endPC geben dabei den geschützten Bereich an und handlerPC den Beginn des Behandlungscodes. Wird handlerPC betreten, so liegt dort immer die geworfene Ausnahme auf dem Operanden-Stack. catchType schränkt den zu behandelnden Ausnahmetyp ein (z.B. java.io.IOException). Sollte catchType null sein, so wird jede Ausnahme gefangen. Dem Code-Attribut können weitere Attribute hinzugefügt werden, wie z.B. LineNumberTable (dient nur Debugzwecken).

5.1.3.1. Die Instruktionsliste

Die Instruktionsliste (implementiert durch teajay.bytecode.attributes.code.JVMInstructionList) dient der Verwaltung von Instruktionen und repräsentiert das *code array* [LYBB12, S. 107] des Code-Attributs. Die Instruktionsliste ist Teil des Code-Attributs und kann wie folgt erhalten werden:

```
1  MethodInfo main = ...;
2  JVMInstructionList lst = main.getCode().getInstructionList();
```

Sie ermöglicht das Hinzufügen, Entfernen und Ersetzen von Instruktionen, solange die Liste noch nicht geschrieben wurde. JVMInstructionList besitzt auch Methoden, die es erlauben Konstanten vom Typ int, long, float und double auf den Stack zu legen:

```
1  lst.pushInt(Integer.MAX_VALUE); //benutzt ldc
2  lst.pushInt(0); //benutzt iconst_0
```

Die Methoden entscheiden dabei, abhängig vom Argument, welche Instruktion dafür zum Einsatz kommt: Es gibt z.B. eine Instruktion (iconst_0), um die ganzzahlige Konstante 0 auf den Stack zu legen. Möchte man aber z.B. die Konstante 10 auf den Stack legen, sollte dafür die Instruktion bipush zum Einsatz kommen. Mit den Methoden JVMInstrucionList.add(...) können der Liste beliebige Instruktionen hinzugefügt werden:

```
1  lst.add(new ALoad(0)); //Lädt die erste lokale Variable als Referenztyp
2  Label myLabel = new Label();
3  lst.add(myLabel, new Ldc("ein string")); //legt eine String-Konstante auf den Stack und ↩
       ↪versieht die Instruktion mit einem Label
4  ...
5  lst.add(new Goto(myLabel)); //springt zu der Instruktion auf die myLabel zeigt
6  ...
```

Dieselbe Instanz einer Instruktion darf immer nur einmal und nur in einer einzigen JVMInstructionList vorkommen. Des Weiteren wird die maximale Stackgröße und die Größe der lokalen Variablentabelle automatisch von der Instruktionsliste berechnet. Zur Berechnung der Stackgröße werden Informationen aus der *stack map table* (siehe 5.1.3.2) herangezogen. Daher kann es zu Fehlern kommen, wenn die Einträge in der Tabelle nicht korrekt sind. Die JVM in der Version 7 lehnt aber sowieso Class-Files mit inkorrekten *stack map tables* ab. Sollte die Anzahl der Elemente auf dem Stack während der Prüfung jemals negativ werden, so wird eine Ausnahme geworfen.

Eine leere Instruktionsliste darf nicht geschrieben werden sondern muss mit mindestens einer Instruktion gefüllt sein. Dabei muss eine Methode immer durch ein throw oder *return verlassen werden. Dies gilt auch für void-Methoden. Der Programmfluss darf niemals aus dem *code array* herausfallen.

5.1.3.2. Die *stack map table*

Die *stack map table* [LYBB12, Paragraph 4.7.4] wird durch teajay.bytecode.attributes.-code.StackMapTable implementiert. Ein Eintrag in der *stack map table* beschreibt den Zustand des lokalen Variablenspeichers und Ausführungsstacks an einer bestimmten Stelle im *code array*. In der Tabelle muss für jede Instruktion, die Ziel eines Sprungbefehls ist, ein Eintrag existieren. Ein Eintrag in der *stack map table* bezieht sich immer auf seinen Vorgänger (je nach Typ in mehr oder weniger Aspekten). Um einen Eintrag vorzunehmen kann wie folgt vorgegangen werden:

```
1  ...
2  JVMInstruction ins = lst.add(new ALoad(0));
3  ...
4  StackMapTable table = main.getCode().getStackMapTable();
5  table.addSameFrame(ins); //Besagt, dass sich relativ zum Vorgänger die Typen der lokalen ↩
       ↪Variablen nicht verändert haben und dass der Stack leer ist.
```

Die Reihenfolge der finalen Einträge (*stack map frame* vgl. [LYBB12, S. 110]) in der *stack map table* wird durch die Reihenfolge des Vorkommens der referenzierten

Instruktion in der Instruktionsliste definiert. Es ist daher egal in welcher Reihenfolge die Einträge hinzugefügt werden. Dabei speichert ein Eintrag die Adresse der Instruktion auf den er sich bezieht als Differenz zu der Adresse des vorherigen Eintrags (*offset delta* vgl. [LYBB12, S. 110ff]). Falls es durch ein zu großes *offset delta* dazu kommt, dass der angeforderte Frame-Typ nicht eingesetzt werden kann, wird vom Framework automatisch ein äquivalenter Frame substituiert. Die Berechnung des *offset delta* wird dabei auch vom Framework übernommen.

Der *initial frame* [LYBB12, S. 110] wird automatisch aus der Methodensignatur berechnet und muss nicht explizit angegeben werden. Das heißt auch, dass sich der erste Eintrag auf den impliziten *initial frame* bezieht. *stack map tables* sind seit der JVM-Version 7 (Version 6 teilweise) verpflichtend und erlauben eine bessere bzw. schnellere Verifikation des Class-File durch die JVM [LYBB12, Paragraph 4.10].

5.1.3.3. Besondere Instruktionsklassen

Einige Instruktionen werden vom Framework nicht Eins zu Eins abgebildet Dadurch entsteht eine Zwischensprache: Zum Beispiel steht die Klasse ILoad(int n), je nach Größe des Arguments, für eine Instruktion der Form iload_n ($n \in [0, 3]$), iload ($n \in [4, 255]$) oder wide iload ($n \in [256, 65535]$). Die Instruktionsklasse ILoad erlaubt es also den gesamten Adressraum der lokalen Variablentabelle zu verwenden, um lokale Variablen vom Typ int auf den Stack zu laden. Die Befehle LLoad, FLoad, DLoad und ALoad verhalten sich ähnlich. Der erste Buchstabe des Befehls steht dabei für den Typ auf dem er arbeitet:

I steht für int. Auf Bytecode-Ebene werden auch boolean, byte, short, char als int dargestellt.

F steht für float.

D steht für double.

L steht für long.

A steht für einen Referenztyp, z.B. java.lang.String.

Diesem Benennungsschema folgen viele Instruktionen der JVM, welche für verschiedene Typen verfügbar sind. Im Falle von long und double gilt zu beachten, dass diese auf dem Stack sowie in der lokalen Variablentabelle immer zwei Adressen belegen. Das heißt nach einem dstore_1 ist die nächste benutzbare Adresse in der lokalen Variablentabelle 3.

Es folgt eine Übersicht über die wichtigsten Instruktionen:

Goto : Je nachdem wie groß die Sprungadresse der angesprungenen Instruktion wird, steht Goto für die Instruktion goto oder goto_w. goto akzeptiert 16-Bit Adressen während goto_w 32-Bit adressen erlaubt. Sprungadressen werden immer relativ zur eigenen Position als Byte-Offset in das *code array* angegeben. Sprungadressen sind also vorzeichenbehaftet.

***Store** : Die Instruktionsklassen AStore, IStore, LStore, FStore und DStore verhalten sich ähnlich wie die entsprechenden *Load-Befehle. Sie dienen dazu lokale Variablen in der lokalen Variablentabelle zu speichern.

IConst : Der IConst-Befehl ist eine Abstraktion der JVM-Instruktionen iconst_n ($n \in [-1, 5]$), bipush und sipush und entscheidet selbstständig, welche der Instruktionen genutzt wird. Dieser kann genutzt werden, um ganzzahlige Konstanten im Bereich -32768 bis 32767 auf den Stack zu legen.

Ldc : Die Instruktionsklasse Ldc steht, je nach Größe der Konstantenpooladresse oder Art der Konstante, für ldc, ldc_w bzw. ldc2_w und kann einige Arten von Konstantenpooleinträgen auf den Operanden-Stack laden:

- Zahlkonstanten wie int, long, float, double. Dabei müssen long- und double-Konstanten immer mit ldc2_w geladen werden.
- Zeichenketten (java.lang.String)
- Klassenobjekte wie z.B. System.class (java.lang.Class)
- Methoden-Handles (java.lang.invoke.MethodHandle) und Methoden-Typen (java.lang.invoke.MethodType).

Switch : Die Switch-Instruktionsklasse steht für die Instruktionen lookupswitch bzw. tableswitch und akzeptiert eine Schwelle für den maximal zulässigen Overhead, ab welcher zu lookupswitch gewechselt wird. lookupswitch benutzt binäre Suche, um die Fälle zuzuordnen, während tableswitch eine klassische switch-Tabelle nutzt und die Sprungadressen vorberechnet. Leere Fälle werden im Falle von tableswitch automatisch zum default-Fall umgeleitet. Die Switch Anweisung kann wie folgt genutzt werden:

```
1  Label defaultCase = new Label();
2  Switch sw = new Switch(1.1f, defaultCase);
3  Label caseLabel = new Label();
4  sw.addCase(3, caseLabel);
5  lst.add(sw);
6  ...
```

In dem Beispiel wird ein maximaler Overhead von 10% angegenen. Verbraucht die tableswitch-Tabelle 10% mehr Speicher im *code array* als die lookupswitch-Tabelle, so wird lookupswitch eingesetzt. Ansonsten wird tableswitch benutzt.

wide : Die Instruktion wide (vgl. [LYBB12, S. 560]) ist in der Lage bestimmte Instruktionen zu vergößern. Dazu akzeptiert die wide-Instruktion als ersten statischen Operanden den Opcode von *load, *store oder iinc. Diese akzeptieren dann eine 16-Bit Adresse als weiteren statischen Operanden. iinc akzeptiert noch zusätzlich eine 16-Bit vorzeichenbehaftete Konstante. Jede Instruktionsklasse, für dessen Instruktion es auch eine wide-Variante gibt, wird diese automatisch, falls nötig, einsetzen. Es gibt daher keine Instruktionsklasse Wide um diese explizit zu nutzen.

Return : Die Klasse Return steht für alle auf der JVM verfügbaren *return-Instruktionen. Im Konstruktor von Return kann der gewünschte Typ angegeben werden.

PlaceholderInstruction : Diese Klasse kann der Instruktionsliste hinzugefügt und angesprungen werden, ohne dass zu dem Zeitpunkt die tatsächliche Instruktion bekannt ist. Natürlich muss vor dem Schreiben der Liste dem Platzhalter eine Instruktion zugewiesen worden sein.

Methodenaufrufe : Die Instruktionsklassen InvokeVirtual, InvokeStatic, InvokeDynamic und InvokeInterface dienen dem Aufruf von Methoden. Das Framework berechnet automatisch aus der Methodensignatur wie viele Elemente diese vom Stack entfernen bzw. hinzufügen werden. Die Instruktionen haben folgende Bedeutung (vgl. [LYBB12, S. 472 - 491]):

invokestatic dient dazu, statische Methoden aufzurufen.

invokevirtual ruft eine Methode mithilfe der virtuellen Tabelle auf.

invokeinterface kann dazu genutzt werden, interface-Methoden aufzurufen.

invokespecial führt einen nicht virtuellen Methodenaufruf durch (z.B. werden Konstruktoren immer mit invokespecial aufgerufen). Sollte die Klasse den Zugriffsmodifizierer SUPER gesetzt haben, so konsultiert invokespecial doch die virtuelle Tabelle und sucht selbständig nach einer geeigneten Super-Methode. Dazu muss allerdings eine Superklasse der aktuellen Klasse im MethodRefInfo angegeben werden. Ab dieser beginnt dann die Suche.

invokedynamic erlaubt es die Methodenauflösung in die Laufzeit zu verlagern. Dazu muss eine *bootstrap*-Methode [LYBB12, S. 472ff] spezifiziert werden, welche den Auflösungsprozess übernimmt. Das Ergebnis dieser Auflösung wird dann von der JVM bis auf Weiteres zwischengespeichert.

Das folgende Beispiel zeigt wie sich der Methodenaufruf System.exit(-1) mit dem Framework umsetzen lässt:

```
1  ...
2  lst.pushInt(-1);
3  lst.add(new InvokeStatic(
4    MethodRefInfo.make("java/lang/System", "exit", "void", "int")));
5  ...
```

Vergleichsinstruktionen : Die meisten Implementierungen von Vergleichsinstruktionen akzeptieren einen Vergleichsmodus im Konstruktor. Zum Beispiel steht DCmp, je nach Vergleichsmodus, für die beiden Instruktionen dcmpg und dcmpl. Viele Vergleichsinstruktionen der JVM akzeptieren auch direkt eine Sprungadresse, wie z.B. if_icmp*, welche vom Framework durch die Klasse IfICmp repräsentiert wird. Andere, wie z.B. dcmp*, legen das Ergebnis des Vergleichs auf den Operanden-Stack. Dabei steht 1 für größer, 0 für gleich und -1 für kleiner. Die beiden Varianten dcmpg und dcmpl unterscheiden sich nur in der Behandlung von NaN-Werten (vgl. [LYBB12, S. 394]).

Eine oft genutzte Klasse von Vergleichsinstruktion sind die Instruktionen if*, welche durch die Klasse If0 implementiert werden. Diese fordert als statischen Operanden eine Sprungadresse und holt nur einen Operanden vom Stack. Dieser wird anschließend immer mit 0 verglichen. Nur wenn der Vergleich erfolgreich ist, wird das Sprungziel angesprungen. Ansonsten wird der Programmfluss einfach weitergeführt.

Die Benutzung der Instruktionsklassen wird nun an einem kleinen Beispiel verdeutlicht. Dabei soll folgender Java-Code in Bytecode übersetzt werden:

```
public static void main(String[] args) {
  int a = 1;
  do {
    a++;
  } while (a < 4);
}
```

Erzeugen des Bytecode mithilfe des Frameworks:

```
    MethodInfo main = ...; //typische main-Methode
    JVMInstructionList lst = main.getCode().getInstructionList();
        Label iinc = new Label("loop_begin");
        lst.pushInt(1);
        lst.add(new IStore(1));
        lst.add(iinc, new IInc(1,1));
        lst.add(new ILoad(1));
        lst.pushInt(4);
        lst.add(new IfICmp(IMode.lt, iinc));
        lst.add(new Return(ReturnType.Void));
        FullFrame frame = main.getCode().getStackMapTable().addAppendFrame(iinc.getTarget());
        frame.addLocal(VerificationType.valueOf(int.class));
```

Im nachfolgenden Beispiel ist zu beachten, dass javap die Sprungadressen absolut angibt, hier aber zur Klarheit relative Sprungadressen verwendet werden. Obiges Beispiel erzeugt folgenden Bytecode:

```
  public static void main(String[] args) {
    0: iconst_1      //int Konstante 1 auf den Stack
    1: istore_1      //Speichern von 1 als lokale Variable 1 (a)
    2: iinc    1, 1  //Inkrementiert die lokale Variable a um 1
    5: iload_1       //Lädt die lokale Variable a
    6: iconst_4      //int Konstante 4 auf den Stack
    7: if_icmplt -5   //Springt zur Adresse 7 + (-5) = 2, wenn a < 4
    10: return       //void return
  }
```

An Adresse 2 muss ein *stack map frame* in der *stack map table* angelegt werden. Dieser kann ein *append frame* oder *full frame* sein. Ein *append frame* zeigt an, dass der Operanden-Stack leer ist und fügt dem vorherigen Frame bis zu drei lokale Variablen hinzu. Sollte ein *append frame* der erste Frame in der Tabelle sein, so bezieht er sich auf den impliziten aus den Parametertypen der Methode berechneten *initital frame*:

```
/* initital frame, existiert nur virtuell */
  frame_type = 255 /* full_frame */
  locals = [ class "[Ljava/lang/String;" ] //java.lang.String[] args
  stack = []
/* erster Frame der Tabelle */
  frame_type = 252 /* append */
  offset_delta = 2 //iinc 1, 1
  locals = [ int ] //lokale Variable int a
```

Die Klassen im Paket teajay.bytecode.attributes.code bilden eine Abstraktion der JVM-Instruktionen und sollen einem Nutzer die Codegenerierung vereinfachen. Auch im Falle der Instruktionsklassen muss ein Nutzer sich nicht um die Eintragung von Konstanten in den Konstantenpool kümmern. Die Klassen veranlassen dies, falls nötig, selber.

Für eine vollständige Dokumentation der verschiedenen JVM-Instruktionen siehe [LYBB12, Paragraph 6.5].

5.1.3.4. Dynamische Adressbindung

Im vorigen Abschnitt wurde über die dynamische Natur der Instruktionsklassen berichtet. Dieses Verhalten wird nur dadurch möglich, dass die Berechnung der Adressen der Instruktionen so spät wie möglich erfolgt. Vor dem Schreiben wird auf allen

Instruktionen die Methode boolean validate() aufgerufen. Diese veranlasst die Instruktionsklasse zu prüfen, ob es aktuell notwendig ist, sich an etwaige Änderungen der Adressierung anzupassen und muss, falls sich die Länge der Instruktion dadurch verändert hat, false zurückgeben. Die Verifikationsschleife sieht wie folgt aus:

```
1  private void validateCycle() throws ByteCodeException {
2    locked = true;
3    long cnt = 0;
4    //Setze initiale Adressen
5    for (JVMInstruction instr : instructions) {
6      instr.setAddress(cnt);
7      cnt += instr.getLength();
8    }
9    boolean success;
10   //Verifiziere die Liste
11   do {
12     cnt = 0;
13     success = true;
14     for (JVMInstruction instr : instructions) {
15       instr.setAddress(cnt);
16       success &= instr.validate();
17       cnt += instr.getLength();
18     }
19   } while (!success);
20 }
```

Quellcode 5.5: JVMInstructionList.java

Solange eine der Instruktionsklassen in der Liste beim Aufruf von validate() false zurückgibt, wird die Schleife wiederholt. Ein einfaches Beispiel einer validate()-Methode ist die Methode Goto.validate():

```
1    boolean validate() throws ByteCodeException {
2      long diff = branchTarget.getAddress() - this.getAddress();
3      if (opcode() == OpCode.vm_goto && (diff > Short.MAX_VALUE || diff < Short.MIN_VALUE)) {
4        this.setOpcode(OpCode.goto_w); //Standard opcode ist OpCode.goto
5        return false;
6      } else {
7        return true;
8      }
9    }
```

Quellcode 5.6: Goto.java

Wenn in einem Durchlauf eine der validate()-Methoden false zurückgegeben hat, so ist die gesamte Liste ungültig und der Durchlauf muss wiederholt werden.

5.1.4. HalloWelt-Programm in Bytecode-Assembler

Im folgenden wird gezeigt, wie man mithilfe des Frameworks ein ausführbares HalloWelt-Programm schreiben kann. Dazu wird zuerst ein ClassFile-Objekt erzeugt, welches die Klasse HalloWelt repräsentiert:

```
1 ClassFile halloWelt = new ClassFile("HalloWelt", ClassAccessFlag.PUBLIC,
2   ClassAccessFlag.SUPER);
```

Quellcode 5.7: HalloWeltGenerator.java

Wird beim Erzeugen keine Superklasse angegeben, so wird automatisch java.lang.Object genutzt. Damit das Programm ausführbar ist, wird als nächstes eine main(String[])-Methode erzeugt und halloWelt hinzugefügt:

```
1 //public static void main(String[] args) {...}
2 MethodInfo main = new MethodInfo("main", new MethodDescriptor(FieldDescriptor.getVoid(),
3   FieldDescriptor.fromClass(String[].class)), halloWelt.getThis(),
4     MethodAccessFlag.STATIC, MethodAccessFlag.PUBLIC);
5 halloWelt.addMethod(main);
```

Quellcode 5.8: HalloWeltGenerator.java

Anschließend wird die Anweisung System.out.println("Hallo Welt"); erzeugt:

```
1 JVMInstructionList lst = main.getCode().getInstructionList();
2 lst.add(new GetStatic(FieldRefInfo.make(ClassInfo.fromClass(System.class),
3   "out", FieldDescriptor.fromClass(PrintStream.class))));
4 lst.add(new Ldc("Hallo Welt"));
5 lst.add(new InvokeVirtual(MethodRefInfo.make(ClassInfo.fromClass(PrintStream.class),
6   "println", FieldDescriptor.getVoid(), FieldDescriptor.getObject())));
7 lst.add(new Return()); //void return
```

Quellcode 5.9: HalloWeltGenerator.java

Dazu wird zuerst der Inhalt des statischen Felds System.out auf den Operanden-Stack gelegt und anschließend die Zeichenkette "Hallo Welt". Nun wird die Instanzmethode PrintStream.println(String) aufgerufen. Diese Methode konsumiert die beiden Operanden auf dem Stack, denn Instanzmethoden brauchen als erstes Argument immer eine Referenz auf ihre Instanz. Diese ist jedoch nicht Teil der anzugebenden Signatur. Anschließend muss halloWelt in eine Datei mit dem Namen HalloWelt.class geschrieben werden:

```
1 halloWelt.write(new File("HalloWelt.class"));
```

Quellcode 5.10: HalloWeltGenerator.java

Das Programm kann nun mit dem Befehl

```
1   > java HalloWelt
```

ausgeführt werden. Eine kompilierbare Version dieses Programms findet sich auf der CD-ROM.

5.2. TeaJays Laufzeitumgebung

Zu TeaJays Laufzeitumgebung zählen die Sprach- sowie Standardbibliothek und Werkzeuge welche die dynamische Bindung von Typ zu Implementierung zur Laufzeit realisieren. Um Implementierungen zur Laufzeit laden zu können, kommt die in Java 7 eingeführte Instruktion invokedynamic sowie Javas Reflection-API zum Einsatz. Zudem bietet die Laufzeitumgebung Werkzeuge an, um TeaJay-Typen in Java entwickeln zu können. Die Laufzeitumgebung ermöglicht es auch, mithilfe von teajay.runtime.Run, TeaJay-Typen auszuführen. Dieser Abschnitt soll nicht die gesamte Implementierung der Laufzeitumgebung dokumentieren, sondern konzentriert sich auf technisch interessante Aspekte. Im Anhang findet sich eine Dokumentation der einzelnen Klassen (im javadoc-Stil).

5.2.1. TeaJays Sprachbibliothek

Die Sprachbibliothek enthält alle essenziellen TeaJay-Typen sowie Typen die der Kommunikation mit Java-Klassen dienen. Hier werden nun die Typen teajay.lang.Number, teajay.lang.java.JavaPrimitives und teajay.lang.java.JavaArray vorgestellt:

Number : Der Typ Number ist fest in TeaJay eingebaut und kann nicht durch TeaJays Sprachmittel neu implementiert werden. Er stellt eine rationale Zahl beliebiger Genauigkeit durch Zähler und Nenner dar. Die interne Darstellung ist als unkürzbarer Bruch, welcher sein Vorzeichen immer im Zähler trägt, normiert. Für die Darstellung von Zähler und Nenner kommt java.math.BigInteger zum

Einsatz. Zudem bietet Number Methoden, die es erlauben Number-Objekte aus Java-Primitiven zu erstellen bzw. Number in Java-Primitive zu konvertieren. Diese sind aus Sicht von TeaJay aber nicht vorhanden und werden nur intern genutzt. Die Sichtbarkeit von Methoden kann durch das Setzen eines speziellen Attributs (teajay.util.tjattribute.TeaJayAttribute) gesteuert werden.

Ein großes Problem bei der Realisierung von TeaJay war, dass es nicht möglich ist, benutzerdefinierte Konstanten in den Konstantenpool eines Class-Files einzubetten. Daher musste ein Weg gefunden werden, Objekte vom Typ Number für den Konstantenpool geeignet zu codieren. Gewählt wurde die Darstellung als Zeichenketten, welche im Konstantenpool nur eine maximale Länge von 65535 Zeichen haben dürfen und somit die Größe der im Quelltext erlaubten Zahlliterale technisch begrenzen. Gespeichert wird der unmodifizierte String, welcher vom Lexer als Zahlliteral erkannt wurde. Mithilfe der Methode Number.$valueOf(String) kann diese Zeichenkette in ein entsprechendes Objekt vom Typ Number umgewandelt werden. Diese Methode nutzt einen volatilen Cache, um nicht wiederholt für jedes Literal ein neues Objekt erzeugen zu müssen. Damit wurde für den Typ Number praktisch ein eigener globaler Konstantenpool geschaffen. Ein ähnlicher Cache existiert auch für die Umwandlung von kleinen int-Werten zu Number, da diese intern häufig genutzt wird.

JavaPrimitives : Der Typ JavaPrimitives bietet Methoden zur Konvertierung von Java-Primitiven zu Number und umgekehrt. Die beiden Methoden zur Konvertierung von long sehen wie folgt aus:

```
1  package teajay.lang.java;
2  ...
3  public static long toLong(Number val) {
4    return val.$long_val(); //Wirft Ausnahme, wenn val zu groß für long ist.
5  }
6
7  public static Number toNumber(long val) {
8    return Number.$valueOf(val);
9  }
10 ...
```

Quellcode 5.11: JavaPrimitives.java

Auch wenn es in TeaJay den Typ long nicht gibt, ist dem Compiler die Existenz durchaus bewusst und erlaubt beispielsweise Aufrufe folgender Art:

```
1      //Java-Code
2      class JavaTest {
3        static long longMethod() { return 4; }
4        static void longMethod(long a) { System.err.println(a); }
5      }
6      //TeaJay-Code
7      Number a = JavaPrimitives.toNumber(JavaTest.longMethod());
8      JavaTest.longMethod(JavaPrimitives.toLong(a));
```

Damit ist es möglich, auch mit Java-Klassen zu kommunizieren die Java-Primitive verwenden.

JavaArray : Die Klasse JavaArray ist abstrakt und bildet die Basisfunktionalität um Java-Arrays in TeaJay nutzen zu können. Direkte Unterklassen von JavaArray sind PrimitiveArray und ReferenceArray. Von PrimitiveArray gibt es für jeden primitiven Java-Typen eine weitere Unterklasse. Diese Arraytypen nutzen die Reflection-API um Java-Arrays zu manipulieren und einen großen Teil der

Typprüfungen in die Laufzeit zu verlagern. Dies ist nötig, da TeaJay keine Java-Arrays kennt.

Einige Typen der TeaJay-Sprachbibliothek sind identisch mit Java-Typen. Dies vereinfacht zum einen die Kommunikation mit Java-Klassen und spart Ressourcen für eine aufwendige Dekodierung von Literalen (wie es bei Number notwendig ist):

java.lang.String : TeaJay nutzt zur Darstellung von Zeichenketten, genau wie Java, den Typ java.lang.String. Alle Methoden die aus TeaJay nicht ohne weiteres aufrufbar sind, werden auf entsprechende Methoden der Laufzeitumgebung umgeleitet. Im Falle von String werden einige Aufrufe zu teajay.runtime.StringType umgeleitet. Diese Klasse ist in TeaJay nicht sichtbar und besitzt nur statische Methoden:

```
1  package teajay.runtime;
2  ...
3  @TJType(type = TeaJayAttribute.Type.Hidden)
4  public final class StringType {
5    ...
6    public static String charAt(String str, Number index) {
7      $_checkIndices(index); //Wirft eine Ausnahme, wenn index keine ganze Zahl ist oder ↩
           ↪nicht in einen int passt.
8      return String.valueOf(str.charAt(index.$int_val())).intern();
9    }
10   ...
11   public static Boolean operator_less(String a, String b) {
12     return a.compareTo(b) < 0;
13   }
14   ...
15   public static Number length(String str) {
16     return Number.$valueOf(str.length());
17   }
18   ...
19   public static String substring(String str, Number beginIndex, Number endIndex) {
20     $_checkIndices(beginIndex, endIndex);
21     return str.substring(beginIndex.$int_val(), endIndex.$int_val());
22   }
23 }
```

Quellcode 5.12: StringType.java

Zum Beispiel wird anstelle der Instanzmethode String.charAt(int) die Methode StringType.charAt(String, Number) aufgerufen. Alternativ hätte man für TeaJay einen eigenen Typ für Zeichenketten erstellen können, jedoch wäre dies mit noch mehr Overhead verbunden und man stünde vor dem gleichen Problem wie bei Number: Es ist nicht möglich, benutzerdefinierte Typen im Konstantenpool unterzubringen.

java.lang.Boolean : Zur Darstellung von Wahrheitswerten nutzt TeaJay den Typ java.lang.Boolean. Für diesen Typ werden keine Methoden umgeleitet, jedoch greift eine Aufrufkonvention des Compilers: Hat eine Methode den Rückgabewert boolean so wird er automatisch zu java.lang.Boolean umgewandelt.

java.io.File und java.lang.System : Die Java-Typen java.io.File und java.lang.System wurden genau wie String durch Umleiten von Methodenaufrufen zu einem eingebauten TeaJay-Typ. Dadurch sind z.B. folgende Aufrufe in TeaJay möglich:

```
1  Number time = System.currentTimeMillis();
2  Number filesize = new java.io.File("somefile").length();
3  System.exit(0);
```

Die o.g. Typen sind zwar TeaJay-Typen, jedoch lässt sich ihre Implementierung nicht ersetzen und sie erben teilweise nicht von teajay.lang.TJObject. TJObject dient nur

als Kompatibilitätsschicht zwischen Java und TeaJay und erlaubt es, die Methode Object.hashCode() indirekt durch Überschreiben von TJObject.tjHashCode() zu überschreiben. Des Weiteren sorgt TJObject dafür, dass die Methode Object.equals(Object) sich genauso wie der TeaJay-Operator == verhält. Damit ist es z.B. möglich, einen TeaJay-Typ, der den Vergleichsoperator überschreibt, als Schlüssel in einer java.util.HashMap sinnvoll einzusetzen.

5.2.2. Binden von Implementierungen an Typen

Ein erstes Binden von Implementierungen und Typen findet bereits beim Initialisieren der Laufzeitumgebung statt. Die dort erzeugten Bindungen werden i.d.R. nicht mehr verändert. Ist jedoch die Kommandozeilenoption -enable-rir aktiviert, so ist es möglich Bindungen auch nachträglich zu verändern. Anders als die Typdefinitionen, werden die Implementierungen erst tatsächlich geladen, wenn sie das erste Mal angefordert werden. Erst dann wird auch klar, ob es die Implementierung überhaupt gibt bzw. sie einsetzbar ist. Dies soll verhindern, dass der Klassenspeicher der JVM zu stark belastet wird und bei vielen vorhandenen Implementierungen im Suchpfad diese den Klassenspeicher überlaufen lassen.

5.2.2.1. Suche nach Implemetierungen

In 3.4.3.8 wurde bereits erläutert wie Implementierungen gesucht bzw. gefunden werden. Die Implementierung der Suche ist recht einfach gehalten und soll vor allem schnell und mit wenig Overhead ablaufen. Aus diesem Grund wurden die .tjt-Dateien eingeführt, denn sie erlauben es, in tiefen Verzeichnisstrukturen, mit potenziell vielen .class-Dateien Implementierungen schnell zu finden. Eine alternative Implementierung könnte z.B. alle .class-Dateien daraufhin prüfen, ob sie Implementierung eines TeaJay-Typs sind. Dazu hätte man entweder das ByteCode-Framework einsetzen können oder jede der .class-Dateien von der JVM laden lassen um sie mit der Reflection-API zu überprüfen. Beide Ansätze wurden getestet und haben entweder zu einer massiven Startverzögerung geführt oder den Klassenspeicher der JVM überlaufen lassen. Durch intelligentere Auswahl der gescannten Pfade hätte man diese Probleme reduzieren können. Der gewählte Ansatz, über die .tjt-Dateien, ist allerdings am einfachsten und schnellsten. Die Implementierung der Suche befindet sich in teajay.runtime.TypeManager.

Bemerkung: Da tj das aktuelle Arbeitsverzeichnis standardmäßig als Klassenpfad nutzt, sollte tj nicht in riesigen Verzeichnissen ausgeführt werden, da diese dann vollständig nach .tjt-Dateien abgesucht werden. Gleiches gilt für den TeaJay-Compiler: Dieser nutzt eine ähnliche Infrastruktur, um nach .class-Dateien zu suchen und indiziert das gesamte Verzeichnis.

5.2.2.2. invokedynamic

Die invokedynamic-Instruktion erlaubt es, Einfluss auf den Auflösungsprozess für Methoden zu nehmen. Eine Methode die mit invokedynamic aufgerufen wird, wird erst zur Laufzeit durch benutzerdefinierten Code aufgelöst. Alle anderen Instruktionen

der JVM erfordern, dass die referenzierte Methode zur Startzeit bereits auflösbar ist. invokedynamic erlaubt es eine vorerst unbekannte Methode einer unbekannten Klasse zu referenzieren. Dabei muss lediglich eine Signatur der Methode bekannt sein und eine Methode, welche die Auflösung übernimmt (eine sog. Bootstrap-Methode). Die Bootstrap-Methode wird bei der ersten Ausführung einer invokedynamic-Instruktion ausgeführt und erzeugt ein CallSite-Objekt, welches eine Bindung zu einer konkreten Methode enthält. Diese Bindung wird zwischengespeichert und bei weiteren Ausführungen derselben Instruktion wird die Bootstrap-Methode vorerst nicht noch einmal ausgeführt. Je nachdem welche Unterklasse von CallSite genutzt wurde, kann die Bindung im Programmverlauf verändert werden.

Eine vereinfachte Version von TeaJays Bootstrap-Methode für Konstruktoren sieht wie folgt aus:

```
1  package teajay.runtime;
2  import java.lang.invoke.*;
3  ...
4  public static CallSite bc(MethodHandles.Lookup callerClass, String name, MethodType mtype, ↩
       ↪Class<?> target) throws Exception, TeaJayResolveError {
5    int mod = target.getModifiers();
6    MethodType voidType = mtype.changeReturnType(void.class);
7    Class<?> impl = target;
8    if (Modifier.isAbstract(mod) || Modifier.isInterface(mod)) {
9      impl = tm.getImpl(target);
10   }
11   return new ConstantCallSite(callerClass.findConstructor(impl, voidType).asType(mtype));
12 }
```

Quellcode 5.13: Bootstraps.java

Die Methode nutzt den TypeManager, um für abstrakte Klassen oder interfaces nach einer Implementierung zu suchen. Sollte die zu erzeugende Klasse instanzierbar sein, so wird dieser Schritt übersprungen und die Klasse direkt instanziiert. Dies vereinfacht die Codegenerierung, da für Java-Klassen und TeaJay-Typen dieselbe Instruktion genutzt werden kann. Um nun mit invokedynamic einen Konstruktor aufrufen zu können, darf nicht die tatsächliche Signatur des Konstruktors verwendet werden: Die Methode hat dieselbe Parameterliste, gibt aber eine Referenz auf das neu erzeugte Objekt zurück (normalerweise haben Konstruktoren keinen Rückgabewert). Die Methode MethodHandle.asType(MethodType) sorgt für eine entsprechende Adaptierung.

TeaJay benutzt noch eine weitere Bootstrap-Methode, um statische Methoden aufzurufen:

```
1  public static CallSite st(MethodHandles.Lookup callerClass, String name, MethodType mtype, ↩
       ↪Class<?> target) throws Exception, TeaJayResolveError {
2    return new ConstantCallSite(callerClass.findStatic(tm.getImpl(target), name, mtype));
3  }
```

Quellcode 5.14: Bootstraps.java

Damit werden statische Methodenaufrufe auf ihre entsprechenden Implementierungen umgeleitet. Für virtuelle Methoden ist dies nicht nötig, da sie die abstrakten Methoden der Typdefinition überschreiben und daher direkt mit invokevirtual bzw. invokeinterface aufrufbar sind.

Folgendes Beispiel zeigt, wie mithilfe des Bytecode-Frameworks eine invokedynamic-Instruktion erzeugt werden kann, welche eine Instanz von List erzeugt. Dazu muss dem ClassFile zuerst das Attribut BootstrapMethods hinzugefügt werden:

```
1  import java.lang.invoke.*;
2  ...
3  ClassFile dyn = ...;
```

```
4 BootstrapMethods boot = new BootstrapMethods();
5 dyn.addAttribute(boot);
6 MethodInfo main = ...;
7 dyn.addMethod(main);
8 JVMInstructionList lst = main.getCode().getInstructionList();
```

Danach wird eine Referenz zu der Methode bc(...) als Bootstrap-Methode eingetragen:

```
1 MethodRefInfo ref = MethodRefInfo.make("teajay/runtime/Bootstraps", "bc",
2   /* Rückgabetyp */FieldDescriptor.fromClass(CallSite.class),
3     /* Parameter */FieldDescriptor.fromClass(MethodHandles.Lookup.class),
4       FieldDescriptor.getString(), FieldDescriptor.fromClass(MethodType.class),
5         FieldDescriptor.fromClass(Class.class)
6 );
7 BootstrapMethodEntry bootstrapMethod = boot.addBootstrapMethod(
8   new MethodHandleInfo<>(MethodKind.REF_invokeStatic, ref),
9     /* statisches Argument der Bootstrap-Methode*/ new ClassInfo("teajay/lang/List"));
```

Dazu wird zuerst ein MethodRefInfo erzeugt, welches auf die gewünschte Methode zeigt. Anschließend wird die Methode als Bootstrap-Methode eingetragen. Dabei können nach dem MethodHandleInfo statische Argumente folgen. Dort können alle Argumenttypen vorkommen, die auch ldc laden kann. In diesem Fall wird bc(...) als letztes Argument eine Klassenobjekt übergeben, welches den zu erzeugenden Typ spezifiziert. Um eine invokedynamic-Instruktion erzeugen zu können muss bootstrapMethod bekannt sein:

```
1 lst.add(new AConstNull());
2 FieldDescriptor listDescr = FieldDescriptor.fromString("teajay/lang/List");
3 lst.add(new InvokeDynamic(new InvokeDynamicInfo(bootstrapMethod,
4   new NameAndTypeInfo<>("anyName", new MethodDescriptor(listDescr, listDescr)))));
```

Dabei muss eine Aufrufkonvention für TeaJay-Konstruktoren beachtet werden: Der Konstruktor einer Implementierung akzeptiert als erstes Argument immer eine Referenz auf seine Identität (vgl. 5.2.3). In diesem Fall reicht es null zu übergeben. Der Methodenname "anyName" wird im Falle von Konstruktoraufrufen ignoriert, da hier klar ist, dass der Methodenname "<init>" lauten muss und bc(...) nur mit Konstruktoren umgehen kann. Die Signatur der aufgerufenen Methode wird hier als List anyName(List) angenommen. Es liegt an bc(...) diese Annahme zu erfüllen. Der Laufzeittyp des Resultats der invokedynamic-Instruktion ist nun i.d.R. teajay.lang.impl.ListImpl.

```
1 lst.add(new InvokeVirtual(MethodRefInfo.make("java/lang/Object",
2   "getClass", "java/lang/Class")));
3 lst.add(new InvokeStatic(MethodRefInfo.make(new ClassInfo("teajay/lang/IO"),
4   "println", FieldDescriptor.getVoid(), FieldDescriptor.getObject())));
5   //Ausgabe: class teajay.lang.impl.ListImpl
```

5.2.3. Vererben bzw. Überschreiben von Methoden

Typdefinitionen werden für die JVM zu abstrakten Klassen übersetzt und alle nicht statischen Operationen werden zu abstrakten Methoden dieser Klasse. Statische Methoden und Konstruktoren dienen vorerst als Platzhalter und dürfen nach JVM-Spezifikation nicht abstrakt sein. Implementierungen erben dann von der abstrakten Klasse, welche den implementierten Typ repräsentiert.

Das größte Problem bei der Implementierung von TeaJay war die Vererbung von Methoden umzusetzen. Die JVM nimmt dem Benutzer im Falle direkter Vererbung die Verwaltung einer virtuellen Tabelle ab. Jedoch stehen in TeaJay aus Sicht der JVM nur dir Typdefinitionen in einer direkten Vererbungsbeziehung. Das heißt eine Implementierung, die z.B. einen Typ B implementiert, welcher Untertyp von A

ist, erbt nur die abstrakten Methoden aus A. Das heißt ein Implementierer von B müsste auch alle Methoden aus A implementieren. Möchte ein Implementierer diese Methoden aber eigentlich nicht überschreiben, ist dieses Vorgehen aus seiner Sicht überflüssig. Um das Erben von implementierten Methoden zu erlauben, verwaltet TeaJay praktisch seine eigene virtuelle Tabelle: Eine Implementierung erzeugt immer eine Instanz des Supertyps des Typs, den sie implementiert, und speichert diese in einem Feld. Alle Methodenaufrufe von nicht überschriebenen Methoden sowie Aufrufe von Super-Methoden werden auf die Instanz des Supertyps umgeleitet. Damit der Implementierung des Supertyps aber eventuelle Methodenüberschreibungen bekannt werden, fordert jeder Konstruktor einer Implementierung eine Referenz auf seine tatsächliche Identität und leitet alle Aufrufe von überschreibbaren Methoden automatisch auf diese um. Dies wird am Beispiel des Typs List klar, welcher ein vollwertiger, in Java geschriebener, TeaJay-Typ ist:

Typdefinition :

```
1  package teajay.lang;
2  ...
3  @TJType(type = TeaJayAttribute.Type.Typedef)
4  public abstract class List<T> extends TJObject implements Cloneable, RandomAccess<T>, ↩
       ↪Collection<T> {
5    public List() {}
6    ...
7    public abstract void addAll(TJIterable<? extends T> vals);
8    public abstract void add(T elem);
9    ...
10 }
```

Quellcode 5.15: List.java

Implementierung (schematisch) :

```
1  @TJType(type = TeaJayAttribute.Type.Impl)
2  package teajay.lang.impl;
3  ...
4  public final class ListImpl<T> extends teajay.lang.List<T> {
5    private final List<T> rthis;
6    private T[] array;
7    private final TJObject rsuper;
8    ...
9    public ListImpl(List<T> rthis) {
10     //Der Laufzeittyp von rthis muss nicht ListImpl sein!
11     if (rthis != null) {
12       this.rthis = rthis;
13     } else {
14       this.rthis = this;
15     }
16     this.rsuper = new TJObject(rthis); //Gebe Identität an den Supertypen weiter
17     this.array = (T[]) new Object[16];
18   }
19   ...
20   public void addAll(TJIterable<? extends T> vals) {
21     TJIterator<? extends T> it = vals.iterator();
22     while (it.hasNext()) {
23       //Hier wird nicht this.add(...) aufrufen!
24       rthis.add(it.next());
25     }
26   }
27   //Folgende Methoden sollen von TJObject geerbt werden
28   public Number tjHashCode() {
29     return rsuper.tjHashCode();
30   }
31
32   public boolean equals(Object obj) {
33     return rsuper.equals(obj);
34   }
35   ... //weitere Methoden weiterleiten
36 }
```

Quellcode 5.16: ListImpl.java

Eine Implementierung eines direkten Untertyps von List würde der Implementierung ListImpl also seine Identität im Konstruktor übergeben. Ein Untertyp von List könnte z.B. so aussehen:

```
1  //Typdefinition
2  @TJType(type = TeaJayAttribute.Type.Typedef)
3  public abstract class MyList<T> extends List<T> {
4    public MyList() {}
5    ...
6  }
7  //Implementierung
8  @TJType(type = TeaJayAttribute.Type.Impl)
9  public final class MyListImpl<T> extends MyList<T> {
10   private final MyList<T> rthis;
11   private final List<T> rsuper;
12   ...
13   public MyListImpl(MyList<T> rthis) {
14     if (rthis != null) {
15       this.rthis = rthis;
16     } else {
17       this.rthis = this;
18     }
19     /* Hier würde vom TeaJay-Compiler invokedynamic genutzt. */
20     this.rsuper = TJTypes.instantiate(List.class, true, new Class<?>[] {List.class}, rthis);
21   }
22   public void add(T value) { //überschreibt add(T)
23     System.out.println("my add");
24     rsuper.add(value); //Ruft die Supermethode auf (in TeaJay reicht nat. super.add(value))
25   }
26   ...
27   public T removeAll(T elem) { //Keine Überschreibung in TeaJays Sinne
28     rsuper.removeAll(elem);
29   }
30   public boolean equals(Object obj) {
31     return rsuper.equals(obj);
32   }
33   //Leite alle restlichen Methoden aus List und TJObject wie oben gezeigt um
34   ...
35 }
```

Quellcode 5.17: MyList und MyListImpl

Die Überschreibung der Methode add(T) ist nun für ListImpl.addAll(...) sichtbar. An diesem Beispiel wird auch deutlich, warum TeaJay-Typen nicht von Java-Typen erben können. Es ist nicht möglich, diesen mitzuteilen, welche Identität diese gerade haben. In Abschnitt 5.3.14 wird gezeigt, dass der TeaJay-Compiler o.g. Umleitungen automatisch erzeugt und so die Vererbung für den Benutzer transparent umsetzt. Zudem müssen noch eventuell benötigte Bridge-Methoden erzeugt werden (siehe Abschnitt 5.1.2.5).

Eine Alternative zu o.g. Vorgehen wäre gewesen, eine virtuelle Tabelle in der Laufzeitumgebung zu verwalten, und TeaJay-Methoden nicht über invokevirtual sondern invokedynamic mit einer speziellen Bootstrap-Methode aufzurufen. Diese Vorgehensweise hätte aber das Arbeiten mit TeaJay-Typen in Java verkompliziert, da alle Methoden über die Laufzeitumgebung aufgerufen werden müssten. Bis auf Erzeugung und Vererbung, kann ein TeaJay-Typ in Java momentan wie eine Klasse verwendet werden.

5.2.4. TeaJay-Typen in Java schreiben

Das Wissen über die Interna von TeaJay-Implementierungen und Typen ist essenziell bei der Entwicklung von in Java geschriebenen TeaJay-Typen. Beim Erstellen von in Java geschriebenen TeaJay-Typen sind vor allem die Annotationen teajay.util.TJType und teajay.util.TJSignature von Bedeutung. Diese erlauben es, mit einem entsprechenden Buildscript, Java-Klassen als TeaJay-Typen zu markieren bzw. Methoden und Klassen eine benutzerdefinierte Signatur zu geben.

TJSignature : Diese Annotation erlaubt es, Einfluss auf das Signature-Attribut einer Methode oder Klasse zu nehmen. Dies sollte nur dann notwendig sein, wenn ein in Java geschriebener TeaJay-Typ mit Closures arbeiten möchte, da

TeaJay-Signaturen ansonsten weitestgehend kompatibel mit Java-Signaturen sind. TJSignature sollte ein String übergeben werden, welcher nach den Regeln aus [LYBB12, Paragraph 4.7.9] formatiert ist.

TJType : Die Annotation TJType erlaubt es zu definieren, welche Art von TeaJay-Typ die annotierte Klasse repräsentieren soll. Sinnvoll sind aus Sicht eines Java-Entwicklers dabei nur die Ausprägungen Impl, Typedef und TJBuiltinType. Standardmäßig wird TJBuiltinType angenommen. Des Weiteren erlaubt die Annotation das Setzen eines speziellen TeaJay-Zugriffsmodifizierers:

None : Keine besonderen Zugriffsmodifizierer (nur die Java-Zugriffsmodifizierer gelten)

Final : Markiert einen Typ als final. Dies kann nicht durch Java-Mittel geschehen, da eine Implementierung von der Typdefinition erben muss.

Abstract : Markiert einen Typ als abstrakt und somit als uninstanziierbar. Abstrakte Typen werden noch nicht vollständig vom Prototypcompiler unterstützt.

Hidden : Versteckt einen Typ oder eine Methode vor dem TeaJay-Compiler.

Wird TJType für eine Methode verwendet, so hat nur das Setzen des Zugriffsmodifizierers Hidden einen Einfluss auf diese.

Am Beispiel von teajay.util.Sorter wird die Benutzung der Annotationen deutlich:

```
1  @TJType(type = TeaJayAttribute.Type.Typedef, accessType = TeaJayAttribute.AccessType.Final)
2  public abstract class Sorter extends TJObject {
3    private static final Class<?>[] sort1 = new Class<?>[] {RandomAccess.class};
4    private static final Class<?>[] sort2 = new Class<?>[] {RandomAccess.class, Closure.class};
5    @TJType(accessType = TeaJayAttribute.AccessType.Hidden)
6    public Sorter() {
7    }
8    @TJSignature("<T:Ljava/lang/Object;>(Lteajay/lang/RandomAccess<TT;>;Lteajay/lang/Closure<↩
         ↪Lteajay/lang/Number;TT;TT;>;)V")
9    public static <T> void sort(RandomAccess<T> list, Closure comparator) {
10     TJTypes.invokeStatic(Sorter.class, "sort", sort2, list, comparator);
11   }
12   public static <T extends TJComparable<? super T>> void sort(RandomAccess<T> list) {
13     TJTypes.invokeStatic(Sorter.class, "sort", sort1, list);
14   }
15 }
```

Quellcode 5.18: Sorter.java

Hieran lässt sich auch erkennen, dass statische Methoden besonders behandelt werden müssen. In TeaJay übernimmt dies der Compiler. Die Klasse teajay.util.TJTypes wird in 5.2.4.1 erläutert. Die Annotationen werden nicht direkt vom TeaJay-Compiler geparst sondern müssen vorher mit dem Programm teajay.util.tjattribute.InjectTJAttribute in das Attribut "TeaJay" umgewandelt werden. Für die TeaJay-Standardbibliothek übernimmt dies das Buildscript. Das TeaJay-Attribut ist in teajay.util.tjattribute.TeaJayAttribute implementiert und enthält Informationen, welche spezifisch für TeaJay sind. Dazu zählt u.a. die Information, ob eine Typdefinition als final deklariert wurde und ob es sich z.B. um eine Typdefinition oder Typimplementierung handelt. Jeder TeaJay-Typ muss dieses Attribut in seiner Attributsliste aufweisen.

Eine in Java geschriebene Implementierung eines Typs, von dem geerbt werden kann, muss sich zusätzlich um die in 5.2.3 genannten Weiterleitungen kümmern (nur von Typen, welche als Typedef markiert sind, kann geerbt werden). Des Weiteren

muss dafür gesorgt werden, dass die Implementierung alle Konstruktoren der Typdefinition deklariert und diese als erstes Argument einen Zeiger auf ihre Identität akzeptieren. Auch der Superkonstruktor-Aufruf muss vom Entwickler übernommen werden, wie es in Quellcode 5.16 zu sehen ist (Initialisierung von this.rsuper). Des Weiteren ist zu beachten, dass keine Konstrukte in der öffentlichen Schnittstelle eingesetzt werden, die TeaJays Typsystem fremd sind. Das sind zum Beispiel innere Klassen, primitive Typen, Arrays, der Zugriffsmodifizierer protected, etc..

5.2.4.1. teajay.runtime.TJTypes

Mithilfe von TJTypes ist es möglich, TeaJay-Typen zu instanziieren und statische Methoden eben dieser aufzurufen. Beispielsweise erzeugt der Aufruf

```
1  List<String> result = TJTypes.instantiate(List.class);
2  result.add("The chief cause of problems is solutions. Eric Sevareid");
```

eine leere Liste von Strings. TJTypes und die Werkzeuge aus 5.2.4 bilden auch eine Bibliothek um das Konzept der Trennung von Typ und Implementierung für Java umzusetzen. Die Beispiele machen aber deutlich, dass dies mit viel Aufwand verbunden ist. Zudem findet kaum statische Typprüfung statt und der Benutzer muss sich um viele Details selber kümmern.

5.3. MiniCompiler

MB
MK

Das MiniCompiler-Projekt ist die Realisierung der Grammatik und Semantik von TeaJay und stellt einen Compiler dar. Dieser erzeugt aus einer TeaJay-Quelltextdatei ein JVM-Class-File. Dazu wird durch einen Parser ein Syntaxbaum erzeugt, welcher anschließend traversiert wird. Dabei werden semantische Prüfungen vorgenommen sowie Code für die JVM generiert.

5.3.1. Der Kompilierprozess

MB

Nachdem die an den TeaJay-Compiler übergebenen Dateien vom Parser in abstrakte Syntaxbäume umgewandelt wurden, beginnt ein vierstufiger Prozess zur semantischen Prüfung und zur Generierung der Class-Dateien. Dieser Prozess wird auf alle übergebenen Dateien durchgeführt:

Stufe 0: Ermittelt aus dem abstrakten Syntaxbaum die Paketzugehörigkeit der Typdefinition, des Aufzählungstyps, des Interfaces oder der Typimplementierung. Folgend werden die import-Anweisungen untersucht. Führt der angegebene Import zu einer Datei, die noch nicht kompiliert wurde, so wird diese zur Kompilation beim Compiler eingetragen. Befindet sich in der Datei eine Typdefinition, wird ermittelt, ob sie öffentlich ist und ob das Schlüsselwort final gesetzt ist. Im Fall eines Aufzählungstypen wird nur die Sichtbarkeit abgefragt. Es wird anschließend der Bezeichner bei allen Entitätsarten ermittelt und daraus wird mithilfe der Paketzugehörigkeit der vollqualifizierte Name der Entität erzeugt. Der Name der Entität in der Datei und die Namen der bekannten Entitäten, die durch die import-Anweisungen geladen wurden, werden der Symbolverwaltung übergeben.

Am Ende von Stufe 0 werden alle zur Kompilation eingetragenen imports an den Parser übergeben, daraufhin werden die erzeugten abstrakten Syntaxbäume von Stufe 0 untersucht. Dies wird solange fortgesetzt, bis keine import-Anweisung zur Neueintragung führt.

Stufe 1: Alle abstrakten Syntaxbäume, die durch Stufe 0 entstanden sind, werden in Stufe 1 nacheinander untersucht. In dieser Stufe werden alle verwendeten Typen im abstrakten Syntaxbaum gesucht. Hierbei wird festgestellt, ob die verwendeten Typen zu einer nicht kompilierten Datei führen. Ist dies der Fall, so werden sie wie in Stufe 0 beim Compiler zur Kompilation eingetragen. Die Ermittlung der verwendeten Typen in Stufe 1 ist notwendig, um in Stufe 2 die Class-Dateien für die Typdefinitionen schreiben zu können. Hierfür werden die vollqualifizierten Namen der verwendeten Typen gebraucht.

Die zur Kompilation eingetragenen Typen werden von Stufe 0 und 1 bearbeitet. Dies wird wie in Stufe 0 solange fortgesetzt, bis keine neuen Typen mehr gefunden werden.

Stufe 2: Die Stufen 0 und 1 waren für das Sammeln von Typnamen verantwortlich. Wenn der Prozess hier angekommen ist, so sind alle verwendeten Typen in der zu kompilierenden Datei bekannt. Das bedeutet, ihre vollqualifizierten Namen stehen zur Verfügung. In dieser Stufe werden alle Methoden, Konstruktoren und Felder im Fall von Typimplementierungen im abstrakten Syntaxbaum gesucht. Dabei werden die Köpfe der Methoden, die Konstruktoren und die Felder über das Bytecode-Framework in die Class-Datei geschrieben. Hierbei wird auch geprüft, ob Methoden mit der gleichen Signatur existieren.

Am Ende dieser Stufe sind alle Entitäten erzeugt, die eine Typdefinition, ein Interface oder einen Aufzählungstyp definieren und alle Methoden einer Typimplementierung bekannt.

Stufe 3: Diese Stufe überführt alle abstrakten Syntaxbäume, die eine Typimplementierung repräsentieren, in eine Class-Datei. Das bedeutet, in dieser Stufe werden fast alle semantischen Prüfungen durchgeführt und ausführbarer Code für die JVM generiert.

Für die Verwaltung von Informationen, die während der Kompilierung einer Datei anlaufen, sind die Klassen Stage0Environment, Stage1Environment, Stage2Environment und Stage3Environment verantwortlich. Diese Klassen befinden sich im Paket codegeneration.environment. Jede Instanz dieser Klassen verwaltet die Instanz der darunterliegenden Stufe. So müssen Informationen nicht ein weiteres Mal gesammelt werden. Die Klassen sind wie folgt beschrieben:

Stage0Environment, verwaltet die ClassFile-Instanz, welche während des Kompilierens das Kompilat repräsentiert. Des Weiteren verwaltet sie alle unbekannten Imports und meldet sie dem Kompilierprozess an und verwaltet die Symboltabelle.

Stage1Environment, verwaltet Typvariablen und sammelt alle aufgetretenen unbekannten Typen und meldet sie dem Kompilierprozess an.

Stage2Environment, verwaltet die generischen Parameter, die auftreten sowie eine TypeUtils-Instanz für Typlöschungen und bietet Hilfsmethoden an, um statische Methoden zu erzeugen. Des Weiteren verwaltet sie alle gefundenen Methodendeklarationen. Die generischen Parameter in dieser Stufe erneut zu sammeln und zu verwalten ist notwendig, da zuvor in Stufe 1 nur um das Auffinden weiterer unbekannter Typen ging und die Typvariablen nicht als unbekannter Typ in den Kompilierprozess gelangen dürfen. In Stufe 2 werden hingegen die Typvariablen gelöscht(siehe. Kapitel 3.4.3.7).

Stage3Environment, verwaltet Typvariablen, Umgebungen für Methoden und einen Zustandsautomat. Außerdem bietet sie Hilfsfunktionen an, für das Erzeugen von Feldern, Closures, Instanzen von Typen und Konstruktoraufrufe von Basistypen.

5.3.2. Der Lexer

MK

Um einen Lexer für TeaJay zu entwickeln, wurde zuerst ein Lexer für Java 7 entwickelt, der sich an der lexikographischen Struktur von Java orientiert (vgl. [GJS+12, Paragraph 3]). Die Änderungen, um aus diesem Lexer einen TeaJay-Lexer zu machen, beschränken sich auf das Entfernen und Hinzufügen von Schlüsselwörtern. Der Lexer ist dreistufig ausgeführt und wurde mithilfe des Lexergenerators JFlex (vgl. [KRD13]) erstellt:

Stufe 1 : Die erste Stufe der Analyse wird von lexer.UnicodeFilteredReader implementiert, welcher eher einen gefilterten Strom als einen Lexer darstellt. Dieser Strom ersetzt sämtliche *unicode escapes* [GJS+12, Paragraph 3.3] in der Eingabe durch ihre entsprechenden Unicode-Zeichen und wurde mit JFlex implementiert.

Stufe 2 : In Stufe zwei geschieht die eigentliche lexikalische Analyse, durchgeführt von lexer.Lexer, welcher ebenfalls mit JFlex implementiert wurde. Dieser Lexer filtert seinen Eingabestrom durch lexer.UnicodeFilteredReader.

Stufe 3 : Die dritte Stufe ist ein Wrapper, der es erlaubt, beliebig viele Token vorauszuschauen (lexer.LookAheadLexer). LookAheadLexer erlaubt es auch, durch das interface lexer.Rule, beliebig komplexe Regeln für eine Vorausschau zu nutzen. Dazu unterstützt er, Markierungen im Tokenfluss zu setzen und zu diesen zurückzuspringen. Zudem gibt es einen speziellen Modus um bestimmte Bitshift-Operatoren (>>, >>> usw.) als einzelne Token zu behandeln. Das ist z.B. nötig um Folgendes parsen zu können:

```
1    List<List<String>>
```

TeaJay unterstützt die Bitshift-Operatoren zwar nicht aber der Lexer bietet diese Funktionalität dennoch an, da zuerst ein Parser für die Java-Grammatik entworfen wurde, welcher anschließend zu einem TeaJay-Parser modifiziert wurde. Zudem wird dadurch Raum für zukünftige Weiterentwicklungen geschaffen.

Der Lexergenerator JFlex wurde für das Generieren eines Java/TeaJay-Lexers angepasst, um nur die Zeichen \r und \n bzw. \r\n als Zeilenumbrüche zu zählen[2]. Das Buildscript kompiliert und startet automatisch die angepasste JFlex-Version.

5.3.3. Der Parser

MB

Ein Parser dient der Transformation von Quelltext in eine andere Struktur. Der TeaJay-Parser ist handgeschrieben und wandelt eine Datei nach der Methode des rekursiven Abstiegs in einen Syntaxbaum um. Diese Methode ist ein einfaches Verfahren, um die Linksableitung einer Grammatik zu ermöglichen. Hierbei führen alle nicht-Terminale zu einem Aufruf einer korrespondierenden Methode, welche das nicht-Terminal produziert. Terminale werden bei Übereinstimmung konsumiert und es wird das nächste Token gelesen. Damit eine Grammatik durch den rekursiven Abstieg behandelt werden kann, muss sie in LL(1) sein und damit folgende Bedingungen erfüllen (vgl. S.269 [ASL+08]):

> Eine Grammatik G ist genau dann LL(1), wenn für je zwei Produktionen $A \to \alpha|\beta$ von G gilt:
>
> 1. Aus α und β sind keine Satzformen herleitbar, die beide mit dem gleichen Terminal beginnen.
> 2. α und β können nicht beide zu ϵ abgeleitet werden.
> 3. Wenn ϵ aus α oder β herleitbar ist, dann beginnt keine aus der anderen Produktion herleitbare Satzform mit einem Terminal, welches in einer beliebigen Satzform von G direkt rechts neben A stehen kann.

Die verwendete Grammatik entspricht nicht der in Kapitel 3 beschriebenen. Dort diente die präsentierte Grammatik dem Verständnis der Struktur von TeaJay und denn Sprachelementen, die vom Compiler zu korrektem Code generiert werden. Im Anhang A, befindet sich die genutzte Grammatik, die der Parser implementiert. Sie ist der Java definierten ähnlich, da diese als Ausgangspunkt diente. Der Parser ist nicht generiert, da es den Autoren wichtig war, ein tieferes Verständnis für die entworfene Grammatik zu entwickeln.

Allerdings gibt es einige Probleme bei der genutzten Grammatik, die das Parsen durch den rekursiven Abstieg erschweren. Als Beispiel zur Erklärung dient die Statement-Regel, die hier vereinfacht dargestellt wird:

Statement:
LocalVarDeclaration | StatementExpression *;*

LocalVarDeclaraton:
Type *identifier* [= Expression] *;*

Type:
identifier [TypeArguments] { . *identifier* [TypeArguments] }

[2]Diese drei Zeichen werden von der JLS als Zeilenumbrüche definiert [GJS+12, S. 19]

StatementExpression:
    Expression *;*

Die Expression-Regel beschreibt z.B. Ausdrücke der Form *identifier* { . identifier } *(* Arguments *)* . Dies ist ein Methodenaufruf. Das Problem bei diesen Regeln liegt darin begründet, dass eine lokale Variablendeklaration und ein Methodenaufruf den selben Präfix besitzen können und die Unterscheidung auch nicht durch eine feste Anzahl von Zeichenvorschau getroffen werden kann. Hierdurch kann die Grammatik nicht in LL(1) sein, da die erste Bedingung nicht erfüllt ist. Die Grammatik allerdings durch Linksfaktorisierung in eine LL(1) Grammatik zu transformieren, erschien den Entwicklern unnötig, so wurde der verwendete Lexer in die Lage versetzt, beliebig komplexe Regeln voraus zu schauen.

Eine Regel in einen Knoten des Syntaxbaums zu transformieren, ist durch die Methode des rekursiven Abstiegs besonders einfach. Als Beispiel zur Demonstration der Transformation der Grammatikregeln wird typeswitch ausgewählt. Zur Erinnerung hier die Regel aus Kapitel 3:

typeswitch (identifier *)* *“{“* <typeswitchblocks> *“}“*

Diese Regel wird dann, wie dargestellt in Java implementiert:

```
1  private TypeSwitchNode parseTypeSwitch() throws CompilerException {
2    if(accept(Sym.TYPESWITCH){
3      if(accept(Sym.BRACKET_OPEN)){
4        if(accept(Sym.Identifier){
5          String ident = lastTokenText;
6          if(accept(Sym.BRACKET_CLOSE)){
7            if(accept(Sym.CURLY_OPEN)){
8              TypeSwitchBlocksNode blocks = parseTypeSwitchBlocks();
9              if(blocks != null){
10               if(accept(Sym.CURLY_CLOSE)){
11                 return new TypeSwitchNode(identifier,blocks);
12               }
13             }
14           }
15         }
16       }
17     }
18     throw new CompilerException(...);
19   }
20   return null;
21 }
```

Wie zu erkennen ist, werden die lexikalischen Elemente typeswitch, (,), { und } an die Methode accept übergeben. Diese Methode erwartet eine symbolische Konstante, die ein Terminal repräsentiert. Mithilfe des Lexers wird überprüft, ob der aktuell gelesene Token mit der symbolischen Konstante übereinstimmt. Ist dies der Fall, so wird die Stringrepräsentation des Tokens in lastTokenText geschrieben und der Lexer liest das nächste Token ein. Stimmen die symbolische Konstante und der Token nicht überein, so wird false zurückgegeben. Des Weiteren ist zu erkennen, dass wenn das erste accept fehlschlägt, null zurückgegeben wird. Dies interpretiert der Parser nicht als Fehler, sondern so, dass die Methode die Regel nicht konsumieren konnte. Eine fehlerhafte Verwendung der Grammatik in einer TeaJay-Datei wird immer durch das Werfen einer CompilerException signalisiert.

Jede Regel wird im Parser zur einer eigenen privaten Methode. Nach diesen Erläuterungen ist der Parser wie folgt implementiert:

```
1  class Parser {
2    //Der Lexer
3    private LexerWrapper lex;
4    //Das gelesene Symbol
5      private Symbol lexSymbol;
6      //Das zuletzt gelesene Token
7      private String lastTokenText;
```

```
8
9     public Parser(LexerWrapper lex){...}
10    public SyntaxNode parse() throw CompilerException{
11      lex.yylex();
12      return this.parseCompilationUnit();
13    }
14    private boolean accept(int symbol){
15          if (symbol == lexSymbol.sym) {
16              this.lastTokenText = l.value;
17              this.lexSymbol = lex.yylex();
18              return true;
19          }
20          return false;
21    }
22    private CompilationUnitNode parseCompilationUnit()throws CompilerException{
23      ...
24    }
25    ...
26  }
```

5.3.4. Abstrakter Syntaxbaum

Der abstrakte Syntaxbaum *[ist] die exakte fein-granulare Zwischendarstellung von Quellcode. [Der] Abstrakte Syntaxbaum [enthält] sowohl syntaktische als auch semantische Informationen, die für vielfältige Analysen genutzt werden können [Pin].* Der Parser in TeaJay erzeugt z.B. aus dem folgenden Quelltext den Baum in Abbildung 5.1:

```
1  typedef Alter {
2    Alter(Number tag,Number monat, Number jahr);
3  }
```

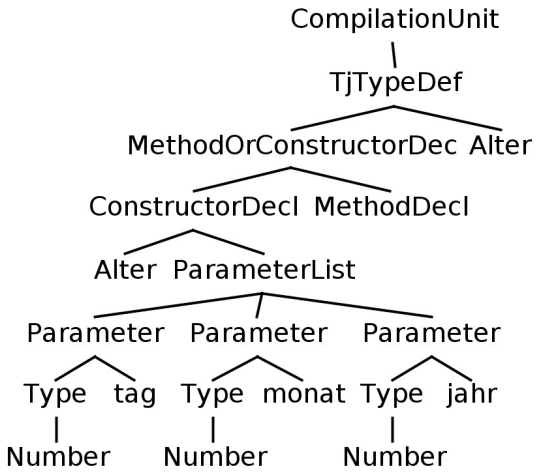

Abbildung 5.1.: Syntaxbaum für Typ Alter

Wie in Abbildung 5.1 zu sehen ist, sind die Blätter des Baums Terminale und die inneren Knoten bilden die syntaktischen Strukturen des Quelltextes ab. Der Vorteil einer solchen Zwischendarstellung des Quelltextes ist, dass er leicht zu traversieren ist.

In TeaJay sind die Knoten des abstrakten Syntaxbaums durch Klassen modelliert. Jede Klasse, die einen konkreten Knoten repräsentiert, erbt von der abstrakten Klasse SyntaxNode, die sich im Paket abstractsyntaxtree befindet:

```
1  public abstract class SyntaxNode {
2      private final Symbol symbol;
3      public SyntaxNode(Symbol s) {
4          this.symbol = s;
5      }
6      public int getLine() {
7          return this.symbol.lineNr;
8      }
9      public Symbol getSymbol() {
10         return this.symbol;
11     }
12     public abstract void accept(Visitor visitor) throws CompilerException;
13   ...
14 }
```

Der Konstruktor erwartet ein Symbol als Parameter, dieser wird während des Parsens vom Lexer generiert. Dadurch ist es möglich, dass Fehlermeldungen erzeugt werden, die einer Zeilennummer und einer Spaltennummer zugeordnet werden können. Das bedeutet, jedes Syntaxelement kennt seine Position im Quelltext. Des Weiteren zwingt die Klasse SyntaxNode ihre Erben dazu, die Methode accept zu implementieren. Dies ist für das Entwurfsmuster Besucher, das für die Traversierung des Baums genutzt wird, wichtig. Damit die semantische Analyse vereinfacht wird, werden während des Parsens und in den Stufen des Kompilierprozesses einige konkrete Syntaxelementklassen um zusätzliche Informationen angereichert. Zum Beispiel, weiß der Expression-Knoten, ob er eine Zuweisung enthält.

5.3.5. Traversierung des Syntaxbaums

Für das Traversieren des Syntaxbaums wird das Entwurfsmuster Besucher verwendet, dass dem Zweck dient *eine auf den Elementen einer Objektstruktur aufzuführende Operation als ein Objekt [zu kapseln]. Das Besuchermuster ermöglicht es, eine neue Operation zu definieren, ohne die Klassen der von ihr bearbeiteten Elemente zu verändern [EG96].* Die Operationen, die auf dem abstrakten Syntaxbaum operieren, sind das Sammeln von Informationen, das semantische Prüfen und die Codegenerierung. Aufgrund des mehrstufigen Prozesses des Kompilierens existieren vier konkrete Besucherhierarchien, die jeweils die Aufgaben, die in Kapitel 5.3.1 genannt sind, durchführen. Der Aufbau des Besuchers unterscheidet sich vom dargestellten Aufbau in [EG96]. Dieser ist nicht flexibel genug für die Objektstruktur, die der Parser erzeugt. Folgende Forderungen setzt das entworfene Muster durch:

- Vor und nach dem Besuch eines Knotens können Operationen durchgeführt werden.
- Exakt eine Art von Knoten kann im Baum besucht werden.
- Um eine Knotenart im Baum zu besuchen soll nicht der gesamte Besucher neu implementiert werden.
- Es wird vorgegeben, welche Methoden überschrieben werden müssen, um einen konkreten Knoten in der Objektstruktur zu besuchen.

Diese Forderungen führten im Entwurf dazu, dass die abstrakte Basisklasse codegeneration.Visitor eingebaut wurde, die für jeden Erben von SyntaxNode drei Methoden begin, visit und end besitzt, jeweils mit dem Erben als Parameter. Somit wird die

erste Forderung erfüllt. Die visit-Methoden der Basisklasse geben immer this zurück[3], während die anderen Methoden keine Operationen durchführen. Hierdurch wird die zweite und dritte Forderung erfüllt, denn durch die triviale Implementierung der visit-Methode kann ein Erbe von Visitor exakt die visit-Methode, für die Knotenart überschreiben, die er benötigt. Die letzte Forderung wird erfüllt, indem für jeden Erben von SyntaxNode eine abstrakte Klasse erstellt wird, die von Visitor erbt und die die geforderten Methoden beinhaltet. Diese Vorgehensweise führt zur Klassenexplosion, hat jedoch den positiven Nebeneffekt, dass Besucherklassen nur noch Verantwortlichkeiten für eine Knotenart des Baums übernehmen.

Der Aufbau der Klassenhierarchie ist verkürzt in Abbildung 5.2 dargestellt. Die Klassen Abstract...Vistor und ...Visitor stehen für die gesamte verbliebene Hierarchie. Eine komplette Darstellung ist an dieser Stelle nicht möglich, da sie zu umfangreich ist.

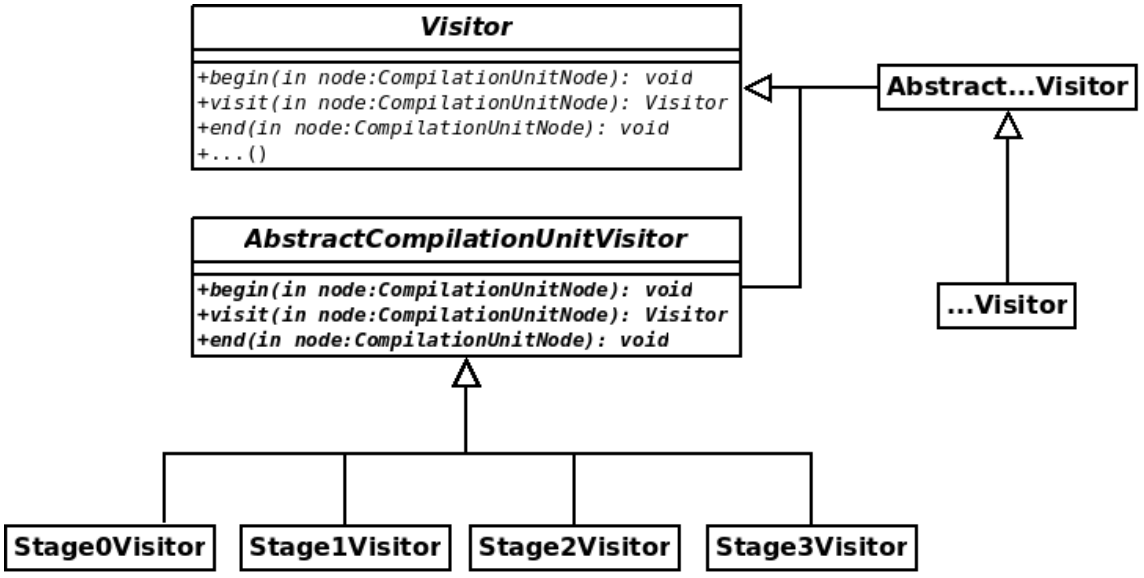

Abbildung 5.2.: Verkürzte Besucherhierarchie

Die Klassen der Besucherhierarchie interagieren mit der Objektstruktur über die accept-Methode der Erben von SyntaxNode. Die accept-Methode erwartet als Parameter einen Visitor übergeben zu bekommen. Durch den Klassenentwurf für das Besuchermuster ergibt sich, dass ein konkreter Besucher für die Kinder eines Knotens des abstrakten Syntaxbaums, aber niemals für den Knoten selbst verantwortlich ist. Eine Ausnahme gilt für die CompilationUnitNode, da dies der Einstiegspunkt der Traversierung ist. Ein Beispiel für eine accept-Methode ist im nachfolgenden Quelltext beschrieben:

```
1  public accept(Visitor visitor) throws CompilerException
2      visitor.begin(this);
3      Visitor v = visitor.visit(this);
4      checkNull(this.assertStatement, v);
5      checkNull(this.block, v);
6      checkNull(this.breakStatement, v);
7      checkNull(this.continueStatement, v);
8      ...
9      visitor.end(this);
10 }
```

[3]Mit der Ausnahme, wenn ein Blatt im Syntaxbaum besucht wird.

Der Quelltext beschreibt die accept-Methode von StatemantNode. Wie zu erkennen ist, kann beim Aufruf von visit ein neuer Besucher zurückgegeben werden. Dieser neue Besucher wird an die Kinder des Knotens über die Hilfsmethoden checkNull weitergegeben. Die Methoden eignen sich für Listen und Referenzen und sind wie folgt in SyntaxNode implementiert:

```
1  protected boolean checkNull(SyntaxNode n, Visitor v) throws CompilerException {
2     if (v == null) {  return true;}
3     if (n != null) {
4         n.accept(v);
5         return false;
6     }
7     return true;
8  }
9  protected boolean checkNull(List<? extends SyntaxNode> list, Visitor v) throws ↩
       ↪CompilerException {
10   if(v == null) { return true;}
11   if(list != null) {
12     for (SyntaxNode s : list) {
13         s.accept(v);
14     }
15     return false;
16   }
17   return true;
18 }
```

5.3.6. Beschreibung von TeaJayUtils

MK

Das Projekt TeaJayUtils enthält folgende Werkzeuge:

teajay.util.JavaNumberParser : Übersetzt sämtliche Zahlliterale, die vom Lexer erkannt werden. teajay.util.JavaNumberParser wird zum Beispiel verwendet, um Zeichenketten aus dem Konstantenpool in Number-Objekte zu konvertieren.

teajay.util.QIDTools : Vereinfacht das Arbeiten mit internen Paketnamen (verwenden "/" als Separator).

teajay.util.TJSignature und teajay.util.TJType : Werden in 5.2.4 beschrieben.

teajay.util.io.* enthält Werkzeuge zum Finden von Ressourcen in Verzeichnisstrukturen und jar-Dateien:

AbstractResourceFinder : Erlaubt das Durchsuchen von Pfaden nach bestimmten Ressourcen. Die Ressourcen werden dabei durch den zum Suchpfad relativen Pfad und den Namen der Datei identifiziert. Als Separator dient dabei unabhängig vom System immer /. Dies entspricht den Namenskonventionen der JVM für interne vollqualifizierte Bezeichner. Ob die gefundenen Ressourcen aus einer jar-Datei stammen oder direkt aus einer Verzeichnisstruktur ist für den Nutzer transparent. Dies geschieht durch das interface Input:

```
1  public interface Input {
2    InputStream openStream() throws IOException;
3    File getFile();
4  }
```

Jeder Ressource wird eine Instanz zugeordnet, welche dieses interface implementiert und in der Lage ist, einen Strom zu öffnen. getFile() darf dabei null zurückgeben (z.B. falls die Datei aus einem jar-Archiv stammt).

ClassFinder : Der ClassFinder dient der Suche und Verwaltung von class-Dateien. Er ist eine Spezialisierung von AbstractResourceFinder. Der Name der vom class-Datei beschriebenen Entität wird dabei aus ihrem Pfad und dem Namen der Datei erschlossen[4]. Dabei werden die class-Dateien faul geladen: Der ClassFinder verwaltet nur Referenzen auf die einzelnen class-Dateien. Die Referenzen verwalten anschließend ein ClassFile-Objekt, welches on-demand erzeugt und zwischengespeichert wird. Es gibt dabei verschiedene Referenztypen, welche alle von der Klasse ClassFileRef erben:

```
1  package teajay.util.io;
2  public abstract class ClassFileRef {
3    public abstract ClassFile get() throws ByteCodeException, IOException, ↩
         ↪ResolveException;
4    public boolean referencesPrimitive() {
5      return false;
6    }
7    public boolean referencesArray() {
8      return false;
9    }
10   public boolean referencesNull() {
11     return false;
12   }
13   public abstract String getQID();
14 }
15 ...
```

Quellcode 5.19: ClassFileRef.java

Standardmäßig wird eine SoftReference (vgl. [Sof]) auf das ClassFile-Objekt verwendet. Diese schützt ihren Referent nur vor der Garbage-Collection solange ausreichend Speicher vorhanden ist. Der ClassFinder wird im Compiler sowie in der Laufzeitumgebung zum Finden von Typen eingesetzt.

TJSourceFinder : Der TJSourceFinder wird vom Compiler eingesetzt, um den Quelltextpfad nach Typen zu durchsuchen, die von anderen Quellcodedateien benötigt werden. Er ist auch ein AbstractResourceFinder.

teajay.util.tjattribute enthält Implementierungen von Attributen (für das ByteCode-Framework), welche der Compiler zum Übersetzen benötigt. Zudem enthält dieses Paket ein Programm, um die in 5.2.4 genannten Annotationen in TeaJay-Attribute zu übersetzen.

teajay.runtime.exceptions stellt einige, häufig benötigte, Ausnahmen zur Verfügung.

FileIterator : Der FileIterator ist ein Werkzeug um Verzeichnisstrukturen nach bestimmten Dateien zu durchsuchen. Dabei ist er auch in der Lage, jar-Dateien zu scannen. Er wird u.a. vom TypeManager eingesetzt, um .tjt-Dateien automatisch zu finden und bildet somit die Grundlage der automatischen Suche nach Implementierungen.

5.3.7. Beschreibung von TypeUtils

Das Paket teajay.generics.check enthält Werkzeuge, die den Umgang mit generischen und parametrisierten Typen sowie Methoden erlauben und stellt somit die

[4]Sollte beim Einlesen der Name, den die Entität deklariert, nicht mit dem Erschlossenen übereinstimmen wird eine Ausnahme geworfen.

Implementierung von TeaJays generischem Typsystem dar. Dazu wurde zum Parsen des Signature-Attributs und zur Darstellung von Typen auf Code des javac (siehe `http://openjdk.java.net/groups/compiler`) zurückgegriffen. Folgende Klassen sind davon betroffen:

- teajay.generics.parser.SignatureParser
- teajay.generics.parser.Type

Die öffentliche Schnittstelle des Pakets wird alleine von der Klasse TypeUtils dargestellt. Diese bietet folgende Funktionalität:

- Die Prüfung der Zuweisungskompatibilität von Typen.
- Die Verwaltung von Typvariablen und die Durchführung von Typlöschung
- Die Überprüfung der Aufrufbarkeit im Falle von Methoden.
- Das Überprüfen von Klassensignaturen und Parametrisierungen auf Gültigkeit.
- Das Erzeugen von Type-Objekten aus Signaturen von Feldern, Klassen und Methoden und umgekehrt.

TypeUtils unterstützt dabei auch den Umgang mit Java-Primitiven und Java-Arrays und setzt die in 3.4.3.6 aufgestellten Regeln zur Zuweisungskompatibilität um. Um beispielsweise die Zuweisungskompatibilität zwischen einer List<String> und List<Object> zu testen, kann TypeUtils wie folgt eingesetzt werden:

```
1  TypeUtils util = new TypeUtils();
2  Type.ClassType lst1 = new Type.ClassType(null, "teajay/lang/List",
3    Arrays.<Type>asList(new Type.ClassType(null, "java/lang/String", null)));
4  Type.ClassType lst2 = new Type.ClassType(null, "teajay/lang/List",
5    Arrays.<Type>asList(new Type.ClassType(null, "java/lang/Object", null)));
6  try {
7    util.checkAssignability(lst1, lst2); //Testet lst1 = lst2
8  } catch (TypeException ex) { //Schlägt fehl
9    System.err.println("not assignable: " + ex.getMessage());
10 }
```

TypeUtils nutzt den ClassFinder, um Typen zu finden. Das heißt bevor TypeUtils zum Einsatz kommen kann, muss der ClassFinder initialisiert sein.

Die verschiedenen Unterklassen von Type sind besuchbar, daher setzt TypeUtils verschiedene Visitoren ein, um seine Aufgaben zu erfüllen. Folgender Visitor erzeugt beispielsweise aus einem Type-Objekt einen FieldDescriptor für das Bytecode-Framework:

```
1  package teajay.generics.check;
2  ...
3  class DescriptorVisitor implements Type.Visitor<FieldDescriptor, Void> {
4      private MethodDescriptor mdescr;
5      public MethodDescriptor getMethodDescriptor() {
6          if (mdescr == null) { throw new IllegalStateException(); }
7          return mdescr;
8      }
9      public FieldDescriptor visitSimpleType(Type.SimpleType type, Void p) {
10         if (type.isPrimitiveType()) {
11             return FieldDescriptor.fromString(type.name);
12         } else {
13             throw new TypeException("expecting erased types");
14         }
15     }
16     public FieldDescriptor visitArrayType(Type.ArrayType type, Void p) {
17         return FieldDescriptor.makeArray(type.elemType.accept(this, null), 1);
18     }
19     public FieldDescriptor visitMethodType(Type.MethodType type, Void p) {
20       ...
```

```
21      }
22      public FieldDescriptor visitClassType(Type.ClassType type, Void p) {
23          FieldDescriptor result;
24          if (type.outerType != null) {
25              FieldDescriptor outerType = type.outerType.accept(this, null);
26              result = FieldDescriptor.fromString(outerType.getClassInfo().getClassName() + "$" +↩
                    ↪ type.name);
27          } else {
28              result = FieldDescriptor.fromString(type.name);
29          }
30          return result;
31      }
32      public FieldDescriptor visitClassSigType(Type.ClassSigType type, Void p) {
33          throw new TypeException("cannot accept class signature here");}
34      public FieldDescriptor visitTypeParamType(Type.TypeParamType type, Void p) {
35          throw new TypeException("expecting erased types");}
36      public FieldDescriptor visitWildcardType(Type.WildcardType type, Void p) {
37          throw new TypeException("expecting erased types");}
38  }
```

Quellcode 5.20: DescriptorVisitor.java

DescriptorVisitor arbeitet nur auf bereits gelöschten Typen. Die Typlöschung wird von einem anderen Visitor erledigt.

Das größte Problem bei der Umsetzung der Typprüfung war zum Einen die schwer verständliche Dokumentation von Javas generischem Typsystem und zum Anderen die Typargumente über Vererbungshierarchien hinweg zu verwalten. Dazu ein Beispiel:

```
1       typedef A<T> {
2         ...
3       }
4       typedef B<T> extends A<List<String>> {
5         ...
6       }
7       typedef C<T> extends B<Number> {
8         ...
9       }
10      A<List<String>> a = new C<String>(); //OK
```

Um die Korrektheit dieser Zuweisung zu prüfen, muss ein Ableitungsbaum aufgebaut werden, welcher auch die Typargumente betrachtet. TypeUtils tut dies in einer indirekten Weise, indem ein spezieller Visitor beginnend von C bis A aufsteigt und die Typargumente substituiert. Dieser ist dann in der Lage die Signatur von C bzgl. A zu erstellen (in diesem Fall A<List<String>>. Anschließend können die Typargumente geprüft werden. Ähnlich werden auch die Signaturen geerbter Methoden bestimmt.

5.3.8. Verwaltung von Namen

MB

In TeaJay haben alle Typen einen Bezeichner. In Kapitel 3.4.2 wurde beschrieben, dass der echte Name eines Typs, der Paketname konkateniert mit dem Bezeichner des Typs ist. Um zu ermöglichen, dass bei der Programmierung in TeaJay die Entwickler mit dem Bezeichner des Typs umgehen können, gibt es eine Namensverwaltung. Durch sie werden Bezeichner von Typen zu ihren vollqualifizierten Namen assoziiert.

Durch das Nutzen der ClassFinder-Klasse ist es möglich, zu ermitteln, ob für einen Typ ein ClassFile-Objekt existiert. Hierdurch lässt sich dann der vollqualifizierte Name eines Typs ermitteln. Es sind folgende Fälle möglich:

- **Importieren eines Typs:**
  Hierbei wird der vollqualifizierte Name eines Typs durch den Entwickler angegeben und somit kann der ClassFinder den Typ anhand des Namens suchen. Dann wird der Bezeichner des Typs mit seinem vollqualifizierten Namen, der Namensverwaltung hinzugefügt.

- **Importieren eines Pakets durch das Wildcard Symbol .*:**
  Der Paketname ist vom Entwickler angegeben und durch den ClassFinder lassen sich alle Typen finden, die den Paketnamen als Präfix im Namen besitzen. Daraufhin werden von allen gefundenen Typen die Bezeichner mit ihren vollqualifizierten Namen der Namensverwaltung hinzugefügt.

- **Typname wird verwendet:**
  Es wird zuerst geprüft, ob die Namensverwaltung den Typnamen kennt. Ist dies nicht der Fall, so wird überprüft, ob der ClassFinder den Namen kennt. Ist auch dies nicht gegeben, so wird angenommen, dass der vollqualifizierte Name des Typs, der Paketname der aktuell bearbeiteten Datei konkateniert mit dem Typnamen ist. Dies wird durch die ClassFinder Klasse geprüft.

Fällt eine der Prüfungen mit der ClassFinder-Klasse negativ aus, so wird mit dem TJSourceFinder geprüft, ob die vollqualifizierten Namen einen Pfad zur einer TeaJay-Datei im System haben. Ist dies der Fall, so werden diese Dateien dem Kompilierprozess hinzugefügt.

Die Verwaltung von Namen übernimmt die symbol.NameResolver-Klasse, die im folgenden Quelltext (verkürzt) gezeigt ist:

```
1  public class NameResolver {
2
3      private final static String[] javaStdLibDefaultImports = {
4          "String", "Boolean", "System", "Object", "Throwable",
5          "Exception", "NullPointerException", "IndexOutOfBoundsException",
6          "IllegalArgumentException", "ArithmeticException", "Error", "IllegalStateException",
7          "RuntimeException", "AssertionError", "Class", "Enum",
8          "NumberFormatException", "OutOfMemoryError", "StackOverflowError",
9          "Runtime", "Thread", "Runnable"
10     };
11     private HashMap<String, String> exisitingNames = new HashMap<>();
12
13     public NameResolver() {
14         ClassFinder finder = ClassFinder.instance();
15
16         for (String s : javaStdLibDefaultImports) {
17             exisitingNames.put(s, "java/lang/" + s);
18         }
19         Map<String, ClassFileRef> res = finder.findAll("teajay/lang");
20         for (String s : res.keySet()) {
21             exisitingNames.put(s, "teajay/lang/" + s);
22         }
23     }
24     public void addNamesFromImport(String qualifiedImportName) throws CompilerException, ↩
           ↪ByteCodeException, IOException, ResolveException {
25         ClassFinder finder = ClassFinder.instance();
26         ClassFileRef findClass = finder.find(qualifiedImportName);
27         if (findClass == null) {
28             throw new CompilerException("Import " + qualifiedImportName + " does not exist");
29         }
30         this.exisitingNames.put(findClass.get().getThis().getSimpleName(), qualifiedImportName)↩
               ↪;
31     }
32     ...
33
34 }
```

Für jede TeaJay-Datei, die kompiliert wird, wird eine Instanz von NameResolver erzeugt. Das Feld javaStdLibDefaultImports deklariert die Namen aus Java, die standardmäßig in TeaJay zur Verfügung stehen. Des Weiteren sind alle Namen aus teajay.lang importiert. Diese beiden Importierungen sind notwendig, damit in TeaJay die Standardtypen und Typen, die eine besondere Bedeutung für die JVM und damit auch für TeaJay besitzen, verwendet werden können.

5.3.9. Lokale Variablentabelle

Das Deklarieren und Verwenden von lokalen Variablen benötigt eine Verwaltung. Wie in Kapitel 3.4.7.3 beschrieben, muss jede Variable deklariert werden, bevor sie genutzt wird. Des Weiteren muss vom Compiler erkannt werden, ob zwei lokale Variablen mit gleichem Namen deklariert werden und die Blockstrukturen einer Methode repräsentiert werden, so dass beim Verlassen eines inneren Methodenblocks die dort deklarierten Variablen vergessen werden. Außerdem muss ein Index verwaltet werden, denn auf Bytecodeebene werden die lokalen Variablen in Registern verwaltet, die mit einem Index adressiert werden.

Diese Aufgaben übernimmt die Klasse LocalTable, die im Folgenden dargestellt ist:

```
1  public final class LocalTable implements Iterable<Frame> {
2
3      private final Deque<Frame> stack = new LinkedList<>();
4      private final StackMapGenerator gen;
5      private final TypeUtils typeUtils;
6      private final MethodInfo methodInfo;
7
8      public LocalTable(MethodInfo info) {...}
9      public LocalTable(MethodInfo methodInfo, TypeUtils typeUtils) {...}
10     public void addFrame() { ...}
11     public void addEntry(String name, Type type) { ...}
12     public void removeFrame() { }
13     public LocalEntry getEntry(String name) { ...}
14     public Iterator<Frame> iterator() { ... }
15     public void addStackMapFrame(JVMInstruction start, VerificationType... stack) { ... }
16     public void addStackMapFrame(JVMInstruction start, List<VerificationType> stack) {...}
17     public int getNextId() { ... }
18     ...
19 }
```

Die symbol.LocalTable hat neben der Aufgabe, die oben beschrieben Verwaltungen und Prüfungen durchzuführen, auch die Aufgabe, die Erstellung von stack map frames anzustoßen. Dies wird durch die Methode addStackMapFrame geleistet. Die Konstruktoren erwarten jeweils ein MethodInfo. Diese ist notwendig, damit die stack map frames der Methode hinzugefügt werden können. Das TypeUtils-Objekt wird beim zweiten Konstruktor erwartet, wenn Typlöschungen durchgeführt werden müssen. Die Methode addFrame wird beim Betreten eines Blocks durch den Besucher und die Methode removeFrame beim Verlassen eines Blocks aufgerufen. Die Methode addEntry fügt dem aktuellen Frame eine lokale Variable hinzu. Hierdurch wird eine lokale Variable einem Block zugeordnet. Als Schlüssel wird der Name der Variablen verwendet. Die Methode getEntry gibt anhand des Namens einer lokalen Variablen ein symbol.LocalEntry zurück. Existiert dieser Name nicht in der Tabelle, wird der Wert null zurückgegeben. Die LocalEntry-Klasse ist wie folgt aufgebaut:

```
1  public class LocalEntry implements Comparable<LocalEntry> {
2
3      private final String qID;
4      private final Deque<String> typeSwitchNameOfTheType = new LinkedList<>();
5      private final Deque<Type> typeSwitchedType = new LinkedList<>();
6      private final int index;
7      private final Type type;
8      private final TypeUtils typeUtils;
9      private final Set<LocalVariableState> variableStates = new HashSet<>();
10
11     LocalEntry(int index, String qID, Type type, TypeUtils typeUtils, LocalVariableState ... ↩
           ↪state) {
12       ...
13     }
14     public int getIndex() {
15        ...
16     }
17     public Type getType() {
18       ...
19     }
20     public String getQID() {
21         ...
22     }
```

```
23      public void setTypeSwitchType(Type t) {
24          ...
25      }
26
27      public void resetTypeSwitchType() {
28          ...
29      }
30      public String getPreviousQID() {
31          ...
32      }
33  }
```

Das Attribut qID ist der vollqualifizierte Name des Typs der Variablen, die eingetragen wird. Aufgrund des Sprachelements typeswitch kann der Typ eines Eintrags wechseln. Dies wird über die Methoden setTypeSwitchType und resetTypeSwitchType sowie die Attribute typeSwitchedType, typeSwitchedNameOfTheType erreicht. Letzterer beinhaltet die vollqualifizierten Namen der Typen. Wird über die Methode setTypeSwitchType einem Eintrag ein Typ hinzugefügt, so geben die Methoden getQID und getType den zuletzt eingetragenen vollqualifizierten Namen und den Typ zurück. Durch resetTypeSwitchType wird dieser entfernt. In Abbildung 5.3 wird ein Beispiel des Zustands der Tabelle, nachdem der Besucher den Methodenkopf der folgenden Methode bearbeitet hat, gezeigt:

```
1  typeimpl TestImpl implements Test{
2  ...
3  void test(String name,Number size){
4    Number a = 2;
5    while(a < 10){
6      Number d = a + 1;
7      if(d.mod( 2 ) == 0){
8        Number z = 1;
9      }else{
10       Number z = 2;
11     }
12     a = d;
13   }
14   Number b = a;
15 }
16 }
```

Local Variable Table

Frame 0

Name	Type	Index	typeSwitchType
this	TestImpl	0	
name	String	1	
size	Number	2	

Abbildung 5.3.: Lokale Variablentabelle, nachdem der Methodenkopf besucht wurde

In Abbildung 5.3 wird deutlich, dass es die lokale Variable this im Frame 0 existiert. Dies liegt darin begründet, dass auf Bytecodeebene im Register 0 immer die this-Referenz verwaltet wird. Aus diesen Gründen haben alle nicht-statischen Methoden eine lokale Variablentabelle, die am Index 0 das this verwaltet. Die Parameter befinden sich wie this in Frame 0. Wird die Variable a in die Tabelle geschrieben, so bekommt die lokale Tabelle keinen neuen Frame, sondern a wird im gleichen Frame wie die Parameter und this abgelegt. Wird das while-Statemant vom Besucher bearbeitet, wird ein neuer Frame angelegt. Diesem wird die Variable d zugeordnet. Dasselbe passiert mit der Variable z im if-Statement, es wird ein neuer Frame angelegt und z wird dort verwaltet. Bevor der if-Block vom Besucher verlassen wird,

Local Variable Table

Frame 0

Name	Type	Index	typeSwitchType
this	TestImpl	0	
name	String	1	
size	Number	2	
a	Number	3	

Frame 1

Name	Type	Index	typeSwitchType
d	Number	4	

Frame 2

Name	Type	Index	typeSwitchType
z	Number	5	

Abbildung 5.4.: Lokale Variablentabelle vor dem Verlassen des if-Blocks durch den Besucher

sieht die Tabelle aus wie in Abbildung 5.4 gezeigt. Hat der Besucher die Bearbeitung des if-Blocks beendet, wird der Frame 2 entfernt und die Indexierung beginnt von 4 an. Ist die Traversierung des while-Blocks vom Besucher durchgeführt, so wird der Frame 1 entfernt und die Variable *b* wird dem Frame 0 zugeordnet.

5.3.10. Berechnung von *stack map frames*

MK

Die Klasse symbol.StackMapGenerator übernimmt die Berechnung von *stack map tables* und nutzt dazu die Informationen der lokalen Variablentabelle. Diese gibt aber keine Auskunft über den Zustand des Operanden-Stacks und daher muss ein Nutzer den Zustand des Stacks explizit angeben. StackMapGenerator berechnet dann die am besten geeignete Art von *stack map table*. Folgende Klassen von *stack map tables* werden von der JVM unterstützt (vgl. [LYBB12, Paragraph 4.7.4]):

same frame : Signalisiert, dass es keine Veränderungen der lokalen Variablentabelle gab und dass der Stack leer ist.

append frame : Kann bis zu drei Variablen an den vorherigen Frame anhängen und signalisiert, dass der Stack leer ist.

chop frame : Kann bis zu drei Variablen vom vorherigen Frame abschneiden und signalisiert, dass der Stack leer ist.

same locals one stack item frame : Signalisiert, dass es keine Veränderungen der lokalen Variablentabelle gab und dass ein Element auf dem Stack liegt.

full frame : Gibt den gesamten Zustand der lokalen Variablentabelle und des Stacks an und bezieht sich dazu nicht auf seinen Vorgänger.

Dabei wird zuerst ein möglicher Frame-Typ nur anhand der Änderungen an der lokalen Variablentabelle berechnet. Anschließend wird anhand der Informationen über den Zustand des Stacks entschieden, welcher Frame tatsächlich zum Einsatz kommen kann.

5.3.11. Der Ableitungsbaum

MB

Der Ableitungsbaum ist neben der TypeUtils-Klasse für das Prüfen der Zuweisungskompatibilität von Typen verantwortlich. Er modelliert die Ableitungshierarchien zwischen Typen. Da in TeaJay ein Typ sowohl von einem anderen Typ wie auch von Interfaces erben kann, ist die Darstellung des Baums verkürzt, wie in Abbildung 5.5 gezeigt. Alle Typen in TeaJay erben indirekt von Object, da TJObject von Object erbt. Am Ableitungsbaum lässt sich die Zuweisungskompatibilität zwischen

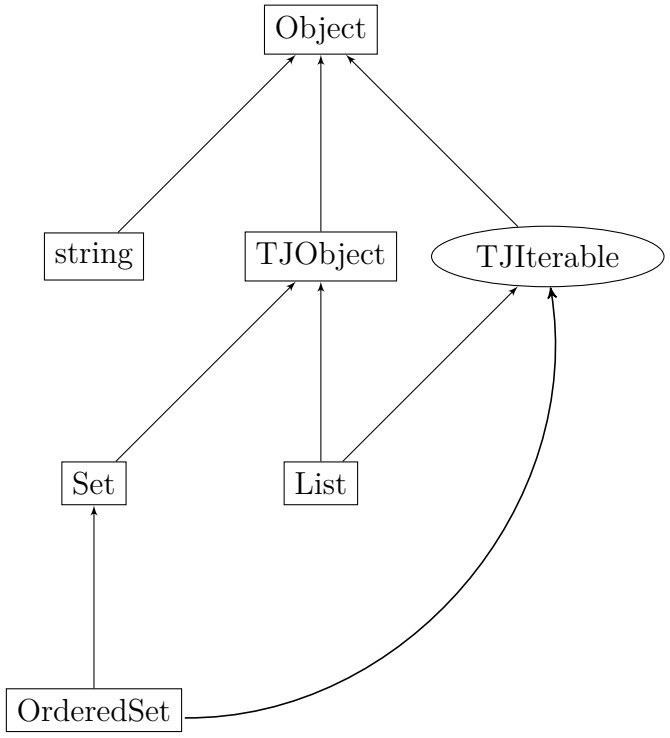

Abbildung 5.5.: Verkürzter Ableitungsbaum

unparametrisierten Typen und Interfaces ablesen. Ein Typ ist zu einem anderen Typ zuweisungskompatibel, wenn die Knoten welche die Typen repräsentieren einen Pfad zueinander besitzen. OrderedSet und List sind beispielsweise zuweisungskompatibel zum Interface TJIterable. Durch die Baumdarstellung ist die Möglichkeit gegeben, Typen von speziell nach allgemein anzuordnen. Dies lässt sich dadurch leisten, dass die Knoten ihre Tiefe im Baum speichern. Allerdings wird die Tiefe zwischen Object und einem Typ nicht durch einen Weg über Interfaceknoten minimiert. Das bedeutet, gibt ein Typ bei seiner Definition ein Interface an, erbt sonst aber nur von TJObject, dann hat er die Tiefe 2.

Der Ableitungsbaum ist in der symbol.DerivationTree-Klasse implementiert:

```
1 public class DerivationTree {
```

```
2       private static DerivationTree instance = null;
3       private final HashMap<ClassFileRef, DerivationNode> map = new HashMap<>(1024);
4       private DerivationNode root; //Object;
5
6       private DerivationTree() throws ByteCodeException, ResolveException, IOException {...}
7       public static DerivationTree instance() throws ByteCodeException, ResolveException, ↩
            ↪IOException {...}
8       public int getDepth(ClassFileRef ref) throws ByteCodeException, IOException, ↩
            ↪ResolveException {...}
9
10      private DerivationNode insertClassDerivation(ClassFileRef ref) throws ByteCodeException, ↩
            ↪IOException, ResolveException {...}
11
12      public boolean isAssignable(ClassFileRef base, ClassFileRef derivate) throws ↩
            ↪ByteCodeException, IOException, ResolveException {...}
13
14      public boolean isDerivated(ClassFileRef base, ClassFileRef derivate) throws ↩
            ↪ByteCodeException, IOException, ResolveException {...}
15  }
```

Die DerivationTree-Klasse ist als Singleton entworfen, da es während des Kompilierprozesses niemals einen weiteren Ableitungsbaum geben kann und an verschiedenen Punkten im Projekt mit ihr kommuniziert werden muss. Die Knoten des Ableitungsbaums werden durch die Klasse verwaltet, indem eine HashMap genutzt wird. Dabei wird eine ClassFileRef zu einer DerivationNode assoziiert. Die HashMap dient der Effizienzsteigerung für Existenzabfragen (Gibt es den Typ im Baum?) und als Sprungtabelle zum geforderten Knoten im Baum. Während des gesamten Kompilierprozesses werden Abfragen bezüglich der Zuweisungskompatibilität gestellt. Ist einer der zu prüfenden Typen noch nicht im Baum repräsentiert, so wird dieser mit seinen geerbten Typen über die Methode insertClassDerivation(ClassFileRef ref) in den Baum eingetragen. Die Methode geht dabei wie folgt vor:

1. Startpunkt des Ableitungspfads ermitteln: Handelt es sich bei ref um einen Typ, wird der Eltertyp ausgewählt. Ist es ein Interface, so wird geprüft, ob das Interface von einem weiteren Interface erbt. Ist dies der Fall, wird das geerbte Interface als Startpunkt des Pfads gewählt. Falls dies nicht so ist, wird Object genutzt.

2. Ableitungspfad aufbauen: Solange der Startpunkt nicht Wurzel des Baums ist, wird der Startpunkt in den Pfad hinzugefügt und ausgehend von diesem Startpunkt wird wie in (1) ein neuer Startpunkt ermittelt.

3. Endpunkt des Pfads in den Baum eintragen: Ist der Endpunkt des Pfads nicht im Baum, wird dem Wurzelknoten der Endpunkt als neuer Kindknoten hinzugefügt und dieser Knoten wird in die HashMap eingetragen. Der neue Knoten hat also den Wurzelknoten als Elterknoten und besitzt die Tiefe 1. Ansonsten wird der Endpunkt aus der HashMap ermittelt.

4. restlichen Pfad in den Baum eintragen: Ausgehend vom erzeugten bzw. ermittelten Knoten wird der restliche Pfad rückwärts durchlaufen. Wenn ein Typ im Pfad noch nicht im Baum eingetragen ist, wird ein neuer Knoten erstellt und als Kindknoten dem ermittelten Knoten hinzugefügt. Ansonsten wird der Knoten des Typs aus der HashMap ermittelt.

5. neuen Knoten erstellen für ref: Nach Schritt vier, ist der Elterknoten von ref im Baum, dann wird ein neuer Knoten erstellt. Dieser bekommt die Tiefe plus eins vom Elterknoten. Er wird als Vater eingetragen und der neu erzeugte Knoten wird der HashMap hinzugefügt.

Die Methode isDerivated(ClassFileRef base,ClassFile derivated) prüft, ob base eine Basis von derivated ist. Die Methode funktioniert wie folgt:

1. Prüfung auf Gleichheit der Referenzen von base und derivated: Im positiven Fall gibt sie wahr zurück.

2. prüfen, ob base und derivated im Baum vorhanden sind: Falls nicht, werden sie darin eingetragen.

3. prüfen, ob die Knoten von base und derivated die gleiche Tiefe haben: Wenn dies zutrifft, wird falsch zurückgegeben.

4. Ausgehend vom Knoten für derivated wird über die Elterreferenz des Knotens geprüft, ob der Knoten der, base repräsentiert, vorkommt. Erreicht die Prüfung die Wurzel des Baums, wird falsch zurückgegeben, ansonsten wahr.

Die Methode isAssignable(ClassFileRef base,ClassFileRef derivated) prüft, ob derivated an base zuweisbar ist. Diese Methode betrachtet die Interfaceknoten mit und funktioniert so:

1. Prüfen, ob derivated von base abgeleitet ist. Ist die Prüfung erfolgreich, dann ist derivated an base zuweisbar.

2. Prüfen, ob base ein Interface ist. Wird dies verneint, so kann derivated an base nicht mehr zugewiesen werden.

3. Ermitteln aller Interfaces, die derivated angegeben hat.

4. Für alle ermittelten Interfaces von derivated wird geprüft, ob base eine Basis vom Interface ist. Ist dies der Fall, dann ist derivated zuweisbar an base.

5. Ist derivated nicht Object, so wird isAssignable mit base und dem Eltertyp von derivated aufgerufen. Ansonsten sind base und derivated nicht zuweisbar.

Der Schritt fünf ist notwendig, da der Eltertyp Interfaces deklarieren kann und somit die Erben auch zu diesem Interface zuweisungskompatibel sind. Die DerivationNode-Klasse repräsentiert die Knoten des Baums und besitzt folgende Schnittstelle:

```
1  class DerivationNode {
2
3      private ClassFileRef classFileRef;
4      private DerivationNode parent;
5      private LinkedList<DerivationNode> childList = new LinkedList<>();
6      private int depth;
7
8      DerivationNode(DerivationNode parent, ClassFileRef classFileRef, int depth) {...}
9
10     public ClassFileRef getClassFileRef() { }
11
12     public DerivationNode getParent() {...}
13     public LinkedList<DerivationNode> getChildList() {...}
14     public int getDepth() {...}
15 }
```

Die Implementierung des Ableitunsgbaums muss mit einigen Spezialfällen, die oben nicht beschrieben wurden, zurechtkommen. Der null-Typ bzw. die null-Referenz hat immer die Tiefe 0 und ist zu allem zuweisbar. Aus Gründen der Kompatibilität zu Java, kann die Klasse mit Java Arrays und den Java Primitiven umgehen. Arrays sind

zuweisbar, wenn auch die Typen zuweisbar sind und ihre Dimensionen gleich sind. Eine Ausnahme gilt, wenn base und derivated Object-Arrays sind, dann ist derivated als Array zuweisungskompatibel, wenn die Dimension von base kleiner gleich der von derivated ist. Primitive Datentypen sind zuweisbar, wenn sie gleiche Datentypen sind, sonst nie.

5.3.12. Methodensuche

MK

Der MethodFinder befindet sich im Paket teajay.methods und bietet Funktionalität, um anhand einer Liste von Typen (Suchliste) eine aufrufbare Methode zu finden. Die Suche geschieht dabei auf gelöschten Typen. Das heißt es muss nachträglich geprüft werden, ob eine eventuell vorhandene Parametrisierung der Typen kompatibel mit der Signatur der Methode ist. MethodFinder.findMethod(...) sucht dabei die Methoden nach dem Best-fit Prinzip (ähnlich wie Java). Dazu wird eine Punktezahl ermittelt, die anzeigt wie gut die Argumente der Methode zur Suchliste passen. Je niedriger die Punktzahl desto besser passt die Methode. Im Falle von Punktgleichheit haben VarArg-Methoden das Nachsehen. Sollte es nach diesen Regeln keine eindeutig beste Methode geben, meldet der MethodFinder dies durch eine Ausnahme. MethodFinder.findAll(...) gibt eine Liste aller Kandidaten zurück. Der MethodFinder gibt zusätzlich zu den gefundenen Methoden (MethodInfo) die erreichte Punktzahl der Methode und die Anzahl an variablen Argumenten zurück (VarArg-Methoden).

Der MethodFinder besitzt verschiedene Modi:

STATIC_ONLY : Sucht nur nach statischen Methoden.

NON_VIRTUAL : Sucht nur direkt in der angegebenen Klasse und steigt nicht die Vererbungshierarchie hinauf.

NON_STATIC : Sucht nur nicht-statische Methoden.

NON_VIRTUAL_NON_STATIC : Kombiniert die Modi NON_STATIC und NON_VIRTUAL.

NON_STATIC_EXACT : Sucht nur nach einer nicht-statischen Methode deren Parameterliste exakt mit der Suchliste übereinstimmt.

ALL : Keine Einschränkungen.

Zur Berechnung der Punktzahl wird folgende Distanzfunktion verwendet:

```
1   function distance(A, B) begin
2     if A ist interface und B ist kein interface
3       return ifaceDistance(B, A, 1)
4     else if B ist null {der Typ null}
5       return 0
6     else
7       return depth(B) - depth(A)
8   end
9
10  function ifaceDistance(B, A, N) begin
11    bestfit = Inf
12    for each interface I of B do
13      if isAssignable(I, A) then
14        tmp = depth(I) - depth(B) + N
15        if tmp < bestfit
16          bestfit = tmp
17      end
18    end
19    for each Supertyp S of A do
```

```
20        if isAssignable(B, S) then
21          tmp = ifaceDist(S, B, N + 1)
22          if tmp < bestfit
23            bestfit = tmp
24        end
25      end
26      return bestfit
27    end
```

Vorbedingung für distance ist, dass B zuweisungskompatibel zu A ist. Die Funktion isAssignable(X, Y) gibt genau dann Wahr zurück, wenn Y zuweisungskompatibel zu X ist, während die Funktion depth(X) die Tiefe von X im Ableitungsbaum zurückgibt. Die Punktezahl ist dann über die Summe der Distanzen der jeweiligen Typen aus der Parameterliste und Suchliste definiert.

5.3.13. Kompilierung von Typedefinitionen

MB

Die Kompilierung einer Typdefinition, betrifft die Stufen 0 bis 2 des Kompilierprozesses. In diesen Stufen werden

- angegebene Modifizierer,
- der Paketname,
- die angegebenen Typen, von denen geerbt bzw. Interfaces implementiert werden sollen,
- generische Typinformationen und
- die spezifizierten Methoden

ermittelt.

Dabei wird eine Typdefinition auf der Ebene der JVM auf eine abstrakte Klasse abgebildet. Hierdurch ergeben sich Konsequenzen, wie die Typdefinition implementiert werden muss. Gibt der Entwickler an, dass der Typ final ist, so darf die Klasse nicht mit dem JVM-Attribut FINAL versehen werden, da es ansonsten unmöglich ist eine Implementierung für den Typen zu erzeugen. Deshalb wird das TeaJay-Attribut Final verwendet. Dieses Attribut wird nur vom Compiler so verstanden, dass von diesem Typ nicht geerbt werden darf.

Generische Typen existieren in der JVM nicht. Das heißt, sie sind vom Compiler zu realisieren. Damit die Generizität eines Typs nach der Kompilierung nicht verloren ist, wird dem erzeugten ClassFile ein Signaturattribut hinzugefügt. Dieses Attribut kann später durch das Bytecode-Framework ausgelesen werden. So wird beispielsweise die Signatur **<T:Ljava/lang/Object;>Lteajay/lang/TJObject;** für **typedef Stack <T>** erzeugt und der ClassFile-Instanz hinzugefügt. Des Weiteren wird jedes Vorkommen der Typvariablen durch die angegebene Typschranke ersetzt.

Jede Klasse auf der Ebene der JVM muss einen Konstruktor besitzen, der den Konstruktor der Elterklasse aufruft. Aus diesem Grund besitzen alle Typdefinitionen einen parameterlosen Konstruktor mit einer Standardimplementierung. Gibt der Entwickler bei der Definition eines Typs keinen parameterlosen Konstruktor an, wird automatisch ein Konstruktor generiert, der als geschützt (protected) markiert ist. So besitzt beispielsweise der Konstruktor Foo()

```
typedef Foo {
  Foo();
}
%
```

diese Implementierung:

```
0: aload 0 // Hole this
1: invokespecial #8 // Method teajay/lang/TJObject."<init>":()V
4: return
```

Dabei gibt die JVM-Spezifikation vor, dass Namen von Konstruktoren <init> lauten (vgl. S.21 [LYBB12]), deshalb wird der Name umgewandelt und in das ClassFile geschrieben. Eine Methode ist im Bytecode-Framework durch die MethodInfo-Klasse repräsentiert und es wird für jede Methode eine Instanz der MethodInfo erzeugt. Hierbei wird der Name der Methode, sein Rückgabetyp, eine Liste der Parameter und die Ausnahmeliste mit angegeben. Sollte die Methode Typvariablen im Rückgabetyp oder den Methodenparameter besitzen, so werden die Typschranken verwendet. Dann bekommen die Methoden Signaturattribute angegeben, welche die Generizität der Methode anzeigen, ähnlich zu der Signatur im o.g. Beispiel. Anschließend werden die nicht-statischen Methoden zusätzlich mit dem Modifizierer abstract markiert und in das ClassFile geschrieben. Die statischen Methoden besitzen eine Implementierung, die das Weiterleiten auf die in einer Typimplementierung implementierte statische Methode realisiert. Diese statische Methode

```
  static Foo operator_add(Foo a, Foo b);
```

besitzt zum Beispiel folgende Implementierung:

```
0: aload_0 //Hole a
1: aload_1 //Hole b
2: invokedynamic #18,  0 // InvokeDynamic #0:operator_add (LFoo;LFoo;)LFoo;
7: areturn
```

Hierbei wird durch den invokedynamic-Aufruf die Laufzeitumgebung von TeaJay angestoßen, nach einer statische Methode zu suchen, die operator_add heißt, zwei Parameter Foo erwartet und einen Rückgabetyp Foo besitzt. Wird die Methode über die Laufzeitumgebung gefunden, so wird sie ausgeführt. Statische Methoden sind nach folgendem Muster aufgebaut:

1. lade alle Parameter aus den Registern auf den Stack der JVM
2. rufe invokedynamic mit dem Namen der Methode, seinen Parametertypen und Rückgabetyp auf.
3. gebe Resultat der Ausführung zurück, wenn die Methode nicht void ist.

Das Umleiten der Methode geschieht, damit alle statischen Methoden durch invokestatic aufgerufen werden können. Hierdurch wird der Umgang mit statischen Methoden erleichtert, da keine Prüfung notwendig ist, ob von einer Typimplementierung, Typdefinition oder eine Java-Klasse eine statische Methode aufgerufen wird.

Zusammengefasst wird eine Typdefinition in TeaJay zu einer abstrakten Klasse umgewandelt. Ihre nicht-statischen Methoden sind alle als abstrakt markiert und die statischen Methoden der Typdefinition leiten ihre Aufrufe an eine Typimplementierung der Typdefinition weiter.

5.3.14. Kompilierung von Typimplementierung

Die Stufen 0 und 1 des Kompilierprozesses für die Typimplementierungen unterscheiden sich nicht stark von denen der Typdefinitionen. Es wird in Stufe 0 nur ein anderer Knoten des abstrakten Syntaxbaums besucht. In Stufe 2 wird, wenn der Kopf der Typimplementierung traversiert wurde, die Typdefinition als Elterklasse in das ClassFile-Objekt geschrieben. Die generischen Parameter einer Typimplementierung werden dem ClassFile, wie bei Typedefinitionen, als Signatur hinzugefügt. Anschließend werden die Signaturen der Typdefinition und der Typimplementierung auf Gleichheit mit der TypeUtils-Klasse geprüft.

Nachdem die Prüfung durchgeführt ist, werden die Methoden- und Konstruktorköpfe der Typimplementierung traversiert. Hierdurch sind alle Methodensignaturen in Stufe 3 bekannt und somit müssen die Methoden vor ihrer Verwendung nicht vorwärts deklariert sein. Die Konstruktoren in einer Typimplementierung bekommen nicht den Namen <init>, sondern ihre Namen lauten $init$. Auf der Ebene des Bytecodes befindet sich nach der Kompilierung in $init$ der Methodenkörper des Konstruktors. Allerdings wird für jeden Konstruktor auch eine <init>Methode mit veränderter Parameterliste generiert. Als ersten Parameter erwartet die <init>Methode eine Referenz der Typdefinition, die implementiert wird.

Das Schema nach dem vorgegangen wird, ist wie folgt: Zunächst werden zwei Felder this und super in der ClassFile-Instanz angelegt. Der Typ von this ist die Typdefinition der Implementierung und der Typ von super ist der Supertyp. Die Implementierung des Konstruktors <init>ist wie folgt beschrieben:

1. Aufruf des Konstruktors der Typdefinition (invokespecial)
2. es wird geprüft, ob das erste Argument null ist. Ist dies der Fall, wird die this-Referenz an das this-Feld übergeben. Andernfalls wird der übergebene Parameter an das this-Feld gebunden.
3. falls vorhanden werden alle restlichen Parameter geladen und die korrespondierende $init$-Methode wird aufgerufen.

In der $init$-Methode wird das super-Feld erzeugt und durch einen invokedynamic-Aufruf initialisiert. Beim Aufruf von invokedynamic wird das this-Feld übergeben. Dies entspricht dem super-Konstruktoraufruf aus Java. Anschließend werden die Instruktionen für den Konstruktorkörper generiert und der InstructionList der MethodInfo-Instanz der $init$-Methode hinzugefügt.

Der Grund für ein solches Vorgehen liegt darin begründet, dass die Vererbung zweier Typimplementierungen nicht von der JVM übernommen werden kann und deshalb durch den Compiler nachgebaut werden muss. Es wird erwartet, dass die Methode eines abgeleiteten Typs die Methode des Basistyps überschreibt. Für die folgende Typhierarchie

```
typedef Base {
  Base();
  //ruft print auf
  void call();
  //Gibt base_call auf der Konsole aus
  void print();
}

typedef Derivated extends Base{
  Derivated();
  //Gibt derivated_call auf der Konsole aus
  void print();
}
```

wird mit folgender Implementierung erwartet,

```
typeimpl BaseImpl implements Base {
  BaseImpl(){ }
  void call(){ this.print(); }
  void print() { IO.println("call_base");}
}

typeimpl DerivatedImpl implements Derivated{
  DerivatedImpl(){ }

  void print(){ IO.println("call_derivated");}
}
```

dass wenn Base base = **new** Derivated() ist, base.call() derivated_call ausgibt.

Für die Vererbung wird das folgende Schema verwendet: Für jede Methode, die in der öffentlichen Schnittstelle des Supertyps definiert ist und nicht in der Typimplementierung des Typs existiert, wird eine Methode mit gleichem Namen und gleichen Parametern in der Implementierung erzeugt und der ClassFile-Instanz hinzugefügt. Folgende Standardimplementierung wird der Methode hinzugefügt:

1. das super-Feld wird geladen
2. alle Parameter der Methode werden aus den Registern geladen
3. die Methode der Basisimplementierung wird über invokevirtual aufgerufen
4. falls ein Rückgabewert vorhanden ist, wird dieser zurückgegeben

Zur besseren Verdeutlichung wird das obige Beispiel in ein Java Äquivalent umgewandelt, allerdings werden die Felder super und this umbenannt, da diese Namen in Java nicht für Felder genutzt werden dürfen. Die Typdefinitionen werden zu diesem Java-Quelltext umgewandelt:

```
public abstract class Base extends TJObject{
  Base(){ }
  //Ruft print auf
  abstract void call();
  //Gibt base_call auf der Konsole aus
  abstract void print();
```

```
}

public abstract Derivated extends Base{
  Derivated() { }
  //Gibt derivated_call auf der Konsole aus
  abstract void print();
}
```

Die Typimplementierungen sehen wie folgt aus:

```
public class BaseImpl extends Base{
  Base thisField;
  Object superField;
  public BaseImpl(Base base){
    super();
    if(base != null){
      thisField = base;
    }else{
      thisField = this;
    }
    $init$();
  }
  public void $init$(){
    //Invokedynamic mit Java nicht möglich
    this.superField = new TJObject(thisField);
  }
  public void call(){
    this.thisField.print();
  }
  public void print(){
    IO.println("base_call");
  }
}
public class DerivatedImpl extends Derivated{
  Derivated thisField;
  Base superField;
  public DerivatedImpl(Derivated derivated){
    super();
    if(derivated != null){
      thisField = derivated;
    }else{
      thisField = this;
    }
    $init$();
  }
  public void $init$(){
    //Invokedynamic mit Java nicht möglich
    this.superField = new BaseImpl(thisField);
  }
  public void call(){
    this.superField.call();
  }
  public void print(){
    IO.println("derivated_call");
  }
}
```

Für dieses Beispiel passiert Folgendes: Der Typ von *base* hat den Laufzeittyp DerivatedImpl. Beim Aufruf der Methode call wird über das super-Feld die call-Methode von BaseImpl aufgerufen. Da das this-Feld den Laufzeittyp DerivatedImpl hat, wird über die call-Methode von BaseImpl die call-Methode von DerivatedImpl aufgerufen.

In Stufe 2 werden die Methodenköpfe einer Typimplementierung im abstrakten Syntaxbaum traversiert. Diese werden, wie bei den Typdefinitionen, in die ClassFile-Instanz geschrieben, mit dem Unterschied, dass sie nicht zusätzlich mit abstract markiert werden. Des Weiteren finden bei statischen Methoden keine Weiterleitung statt. Allerdings müssen beim Hinzufügen von Methoden in das ClassFile Prüfungen durchgeführt werden. Die Methoden, die öffentlich markiert sind, müssen in der Ableitungshierarchie der Typdefinition bzw. in den angegebenen Interfaces existieren. Im Gegensatz dazu dürfen Methoden, die als private markiert sind, dort nicht existierten. Des Weiteren wird geprüft, ob eine Methode mit gleichen Namen und Parametern in der ClassFile-Instanz existiert. Für die Erzeugung und Prüfung ist die Hilfsklasse MethodHeadBuilder verantwortlich. Das Prüfen, ob eine öffentliche Methode der Typimplementierung auch in der Hierarchie definiert wurde, geschieht so:

1. Über die ClassFinder-Klasse wird das ClassFile der Typdefinition der Typimplementierung ermittelt. Dies ist der Iterationsstartpunkt
2. Solange die Iteration nicht java.lang.Object erreicht hat:
   a) Hole alle Methoden des ClassFiles mit dem Namen der zu prüfenden Methode!
   b) Vergleiche die Signaturen der ermittelten Methoden mit der zu prüfenden Methode!
   c) Falls eine Methode gefunden wurde, die in der Signatur übereinstimmt, prüfe die Modifizierer ohne dabei abstract zu beachten!
   d) Falls Übereinstimmungen gefunden wurden, breche Schleife ab! Die Prüfung ist erfolgreich.
   e) Ansonsten wird die Iteration mit der Basis der Typdefinition fortgesetzt.
3. Falls die Schleife durchgelaufen ist, ohne die Methodendefinition zu finden, werfe eine Ausnahme!

Sollte die Prüfung scheitern, so wird sie mit den definierten Interfaces der Typdefinition fortgesetzt. Scheitert auch dies, so wird eine Ausnahme geworfen. Am Ende der Traversierung von TjTypeImplNode wird eine Prüfung durchgeführt, ob alle Methoden der Typdefinition in der Typimplementierung existieren.

5.3.15. Kompilierung von Closures

MK

Die Übersetzung von Closures verläuft ähnlich der Übersetzung von anonymen Implementierungen in Java. Für jedes Vorkommen des Schlüsselwortes closure wird eine neue Klasse erzeugt. Diese erbt von teajay.lang.Closure:

```
@TJType(accessType = TeaJayAttribute.AccessType.Abstract)
public abstract class Closure extends TJObject {
  protected Closure() {}

  @TJType(accessType = TeaJayAttribute.AccessType.Hidden)
  public abstract Object execute(Object... params) throws Throwable;
  ...
}
```

Quellcode 5.21: Closure.java

Der Klasse wird ein eindeutiger Name der Form <Impl>$Closure<$n$> gegeben. Dabei steht n für die bisherige Anzahl an Closures in der aktuellen Implementierung. Die Umgebung des Closures wird ausgewertet und der Klasse über den Konstruktor übergeben. Dieser initialisiert die entsprechenden Felder mit den übergebenen Werten. Beispielweise wird folgendes Closure

```
typeimpl MyImpl implements ... {
  void run() {
    String str = "some string";
    Closure<Boolean>(Number) cl = closure[Number a = 5; String str = str;]<Boolean>(Number n) {
      IO.println(str);
      return n == a;
    };
    cl(7);
  }
}
```

zu einer Klasse der Form

```
@TJType(type = TeaJayAttribute.Type.Closure)
final class MyImpl$Closure0 extends teajay.lang.Closure {
  private Number a;
  private String str;
  public MyImpl$Closure0(Number p1, String p2) {
    a = p1; str = p2;
  }
  Object execute(Object... args) {
    Number n = (Number) args[0];
    {
      IO.println(str);
      return TJObjects.equals(n, a);
    }
  }
}
```

übersetzt. Der Closure-Block wird ähnlich zu einem Methodenblock traversiert. Da MyImpl$Closure0 keine Informationen über die Anzahl bzw. den Typ der Parameter des Closures enthält, wird der statische Typ (Closure<Boolean>(Number)) für Closures getrennt vom tatsächlichen Typ verwaltet. Der tatsächliche Typ wird benötigt, um auf die privaten Felder zugreifen zu können. Die Verwaltung übernimmt die Klasse MethodState, welche Informationen über den statischen Typ vorhält bzw. anzeigt, ob gerade ein Closure traversiert wird. Innerhalb eines Closures hat die this-Referenz den statischen Typ des Closures, was es ermöglicht, dass Closures sich selber rekursiv aufrufen können. Der statische Typ des Closures wird auch in die lokale Variablentabelle, für die Variable cl, der Methode run() eingetragen. Der Typ Closure wird von TeaJays generischem Typsystem als ein besonderer Typ behandelt, welcher beliebig viele Typargumente akzeptiert. Der Typ Closure<Boolean>(Number) wird dabei zu folgender Signatur übersetzt:

```
Closure<Boolean, Number>
```

Diese kann einfach in einer Type-Instanz gespeichert werden und damit in ein Signature-Attribut codiert werden. Die erste Typvariable von Closure steht dabei immer für den Rückgabetyp. Der Typ eines Closures ohne Rückgabewert ist Closure<java.lang.Void, ...>. Daher kann aus einem Closure nie mit einem void-return

zurückgekehrt werden (auf Bytecode-Ebene). Es wird im Falle eines void-Closures immer return null; verwendet. Dies wird auch schon deutlich, wenn man sieht, dass execute(...) Object als Rückgabetyp hat. Um obiges Closure zu erzeugen und auszuführen, wird in der Methode run() zu folgendem Java-Code äquivalenter Code generiert:

```
1 Closure cl = new MyImpl$Closure0(Number.$valueOf("5"), str);
2 cl.execute(Number.$valueOf("7"));
```

Der Konstruktor-Aufruf wird dabei nicht wie in Java über new und invokespecial sondern über invokedynamic realisiert (siehe Abschnitt 5.3.17.5).

Bei der Traversierung eines Closures wird eine neue Umgebung erzeugt, welche jedoch einige Informationen aus der alten übernimmt. Dazu zählen z.B. die Typvariablen. Dies wird dadurch erreicht, dass sich das Closure und die umgebende Implementierung u.a. dieselbe TypeUtils-Instanz teilen.

5.3.16. Kompilierung von Aufzählungstypen

TeaJay-Aufzählungstypen sind kompatibel zu Java-Aufzählungstypen und ihre Erzeugung orientiert sich am Vorgehen des javac. Die Übersetzung eines Aufzählungstyps (enum) wird hier exemplarisch, am Beispiel von myenum.MyEnum, gezeigt:

```
1 package myenum;
2 public enum MyEnum {
3   A, B;
4 }
```

Quellcode 5.22: MyEnum.tj

Zuerst wird dazu eine Klasse Namens MyEnum erzeugt, welche von java.lang.Enum erbt. Die Klasse hat dabei immer die Zugriffsmodifizierer FINAL und ENUM gesetzt. Anschließend wird für jede Konstante ein statisches Feld vom Typ MyEnum, welches den Namen der Konstanten trägt, erzeugt und im static-Konstruktor initialisiert. Folgender Java-Code ist weitestgehend äquivalent zum Bytecode den der TeaJay-Compiler für obigen Aufzählungstypen generiert:

```
1  package myenum;
2  public final /*enum*/ class MyEnum extends Enum<MyEnum>
3    implements teajay.lang.TJComparable<MyEnum> {
4
5    public static /*enum*/ final MyEnum A;
6    public static /*enum*/ final MyEnum B;
7    private static /*synthetic*/ final MyEnum[] $VALUES;
8    static {
9      A = new MyEnum("A", 0); B = new MyEnum("B", 1);
10     $VALUES = new MyEnum[] {A, B};
11   }
12   private MyEnum(String name, int ordinal) {
13     super(name, ordinal);
14   }
15   public static MyEnum[] values() { //Wird von Enum benötigt
16     return $VALUES.clone();
17   }
18   public static MyEnum valueOf(String name) {
19     return Enum.valueOf(MyEnum.class, name);
20   }
21   //TeaJay-Methoden
22   public static teajay.lang.List<MyEnum> getConstants() {
23     return teajay.util.ArrayUtils.toList(this.values());
24   }
25   public teajay.lang.Number tjCompareTo(MyEnum m) {
26     //Die Klasse Enum implementiert das interface Comparable
27     return teajay.lang.Number.$valueOf(this.compareTo(m));
28   }
29 }
```

Aufzählungstypen brauchen den Vergleichsoperator nicht explizit zu überschreiben, da es von jeder Konstante immer nur eine Instanz gibt und die Standardimplementierung von == auf referenzielle Gleichheit prüft. Die Anordnung der Aufzählungstypen (bzgl. TJComparable) wird nach dem Vorkommen im Quelltext bestimmt. Das heißt in dem Beispiel ist $A < B$. Der Zugriff auf Enum-Konstanten ist dann ein lesender statischer Feldzugriff. Ein schreibender Zugriff auf das Feld einer Konstanten wird durch den Zugriffsmodifizierer final verhindert.

5.3.17. Kompilierung von Ausdrücken

MB

Für Ausdrücke wird hier beschrieben, wie sie auf grammatikalischer Ebene erstellt wurden. Die Grammatikregeln für Ausdrücke stammen ursprünglich aus der Java-Spezifikation, sie lassen sich allerdings nicht mit einem Parser in ein Syntaxbaum transformieren, der auf dem rekursiven Abstieg beruht, da sie linksrekursiv sind. Die Grammatik (vgl. S.602f [GJS+12]) sieht wie folgt aus:

Expression:
Expression1 [AssignmentOperator Expression1]

Expression1:
Expression2 [Expression1Rest]

Expression2:
Expression3 [Expression2Rest]

Expression3:
PrefixOp Expression3
| (Expression | Type) Expression3
| Primary { Selector } { PostfixOp }

Die Linksrekursivität besteht hier indirekt. Die folgende Ableitung zeigt sie:

$$\begin{aligned}
&\textit{Expression} \rightarrow \textit{Expression}1[\textit{AssignmentOperator}\ \text{Expression1}] \\
&\rightarrow \textit{Expression}2[\textit{Expression}1\textit{Rest}][\textit{AssignmentOperator}\ \text{Expression1}] \\
&\rightarrow \textit{Expression}3[\textit{Expression}2\textit{Rest}][\textit{Expression}1\textit{Rest}][\textit{AssignmentOperator}\ \text{Expression1}] \\
&\rightarrow \text{Expression Expression3}[\textit{Expression}2\textit{Rest}][\textit{Expression}1\textit{Rest}] \\
&[\textit{AssignmentOperator}\ \text{Expression1}]
\end{aligned}$$

Das Auflösen der indirekten Linksrekursion geschieht über ein Verfahren, welches z.B. in [Moo00] [LN] präsentiert ist:

```
1 Arrange nonterminals in some order A_1, A_2, ..., A_n.
2 for i := 1 to n do begin
3   for j := 1 to i - 1 do begin
4     Replace each production of the form A_i -> A_j β by
5     the productions:
6         A_i → α_1β|α_2β|...|α_kβ
7     where
8         A_j → α_1|α_2|...|α_k
```

```
9       are all the current A_j productions.
10    end
11    Remove immediate left recursion from the A_i productions, if ↩
          ↪necessary.
12  end
```

Durch das Anwenden des obigen Algorithmus ergibt sich folgende Zwischentransformation:

Expression:
Expression1 [AssignmentOperator Expression1]

Expression1:
Expression2 [Expression1Rest]

Expression2:
Expression3 [Expression2Rest]

Expression3:
Expression3 [Expression2Rest] [Expression1Rest] [AssignmentOperator Expression1] Expression3
| PrefixOp Expression3 | Type Expression3 | Primary { Selector } { PostfixOp }

Diese beinhaltet nur noch eine direkte Linksrekursion, die sich durch dieses Verfahren in eine rechtsrekursive Form transformieren lässt: Ersetze jede Regel der Form

$$A \to A\alpha|\beta$$

durch

$$A \to \beta A_1$$
$$A_1 \to \alpha A_1|\epsilon$$

Durch das Auflösen der Linksrekursion und die Eliminierung von Regeln, die für TeaJay ohne Bedeutung sind, ließ sich die Grammatikregel dazu vereinfachen:

Expression:
Expression1 [AssignmentOperator Expression1]

Expression1:
Expression2 [Expression1Rest]

Expression2:
Expression3 [Expression2Rest]

Expression3:
PrefixOp Expression3 Exp3Rest
| Primary { Selector } { PostfixOp } Exp3Rest

Exp3Rest:

[Expression2Rest] [Expression1Rest] [AssignmentOperator Expression1] Expression3 Exp3Rest | ϵ

Dass die Regel Type Expression3, wie oben gezeigt, entfernt wurde, ist dadurch begründet, dass durch sie kein sinnvoller Ausdruck für TeaJay erzeugt wird. Weiterhin ist festzuhalten, dass das nicht-Terminal Expression3 in Exp3Rest zu unerwünschten Ausdrücken führt, wie z.B. $1 + 2\ 3$. Ein Entfernen der Regel würde allerdings dazu führen, dass Exp3Rest selbst wieder linksrekursiv ist.

Da die optionalen Regeln [Expression2Rest] (siehe, Kapitel 5.3.17.2) und [AssignmentOperator Expression1] in Exp3Rest ausreichen, um für TeaJay alle sinnvollen Zuweisungs- und arithmetischen Ausdrücke zu produzieren, wurde aus Exp3Rest die Expression3 und die Exp3Rest Produktion entfernt. Des Weiteren wurde jedes Vorkommen von Exp3Rest in Expression3 durch ihre Produktion ersetzt. Daraus ergibt sich, dass die folgende Grammatik genutzt und implementiert wurde. Zur besseren Darstellung existiert in der folgenden Grammatik die Exp3Rest-Regel:

Expression:
Expression1 [AssignmentOperator Expression1]

Expression1:
Expression2 [Expression1Rest]

Expression2:
Expression3 [Expression2Rest]

Expression3:
PrefixOp Expression3 Exp3Rest
| Primary { Selector } { PostfixOp } Exp3Rest

Exp3Rest:
[Expression2Rest] [Expression1Rest] [AssignmentOperator Expression1]

Eine Aufgabe bei der Generierung von Ausdrücken ist es, den Laufzeitstack der JVM in einem statischen Modell des Compilers zu verwalten. Dieses statische Modell, es wird von nun an Typstack genannt, ist essentiell für die Gewährleistung der statischen Typsicherheit. Es ist festzuhalten, dass der Typstack nur bis zu einem gewissen Grad mit dem Laufzeitstack übereinstimmt. Beispielsweise besitzt die Methode test

```
1 static Object getString(String t){
2   return t;
3 }
4 void test(){
5   Object obj = getString("test");
6 }
```

mit dieser Bytecodedarstellung

```
1 0: ldc hallo
2 2: invokestatic getString:(Object;)Object;
```

```
3 5: astore 1
```

den hier beschriebenen Laufzeit und Typstack:

Laufzeitstack
Der Laufzeitstack ist zu Beginn leer. Durch die Instruktion ldc wird ein String auf den Stack gelegt. Die Instruktion invokestatic konsumiert den String und legt einen String auf den Laufzeitstack zurück. Am Ende wird durch die Instruktion astore der String in das Register 1 gespeichert und der Laufzeitstack ist wieder leer.

Typstack
Zu Beginn wird ein String auf den leeren Stack gelegt, dann ein String konsumiert, aber ein Object auf den Stack zurückgelegt. Am Ende wird vom Typstack ein Object konsumiert und der Typstack ist wieder leer.

Der Typstack wird in der ExpressionEnvironment-Klasse im Paket codegeneration.environment verwaltet. Sie bietet eine Reihe von Hilfsmethoden an, um Ausdrücke zu generieren und den Typstack zu verwalten. Die Wichtigsten für die Verwaltung sind

void setResultType(Type)
fügt den Typ als oberstes Element dem Stack hinzu.

popType()
entfernt das oberste Element vom Stack und gibt es zurück.

getOperands()
gibt den aktuellen Stack als Liste zurück. Die Funktion wird benötigt, um stack map frames in den Fällen in denen eine Berechnung durch den StackMapGenerator nicht möglich ist, zu erzeugen

Ein weiterer Aspekt bei der Generierung von Ausdrücken ist die Unterscheidung zwischen lesendem und schreibendem Zugriff auf Variablen. Im Allgemeinen kann davon ausgegangen werden, dass alle Ausdrücke die links von einem Zuweisungsoperator stehen als L-Wert behandelt werden müssen und alle rechts davon als R-Wert. Diese Unterscheidung wird bei der Transformation eines Ausdrucks zu einem Teilbaum des abstrakten Syntaxbaums getroffen. Bei der Generierung ist darauf zu achten, dass der Ausdruck links neben einem Zuweisungsoperator erst generiert werden darf, wenn die rechte Seite generiert wurde. Die rechte Seite des Ausdrucks ist generiert, wenn die Expression1 in der optionalen Grammatikregel AssignmentOperator Expression1 von Expression3 traversiert wurde. Erst dann kann die linke Seite des Ausdrucks generiert werden. Für diese Aufgabe stehen die Klassen VariableInstructionCreator mit ihren zwei Kindern RValueInstructionCreator und RValueInstructionCreator, die sich im Paket codegeneration.instruction befinden, zur Verfügung. Diese bieten Methoden an, um lesende bzw. schreibende Zugriffe auf Variablen zu generieren:

createClassInstruction(...)
ermöglicht den Zugriff nur auf statische Felder über den Namen eines Typs. Aus Kompatibilitätsgründen kann diese Methode auf öffentliche statische Felder von Java-Klassen zugreifen.

createFieldInstruction(...)
generiert den Zugriff auf Felder. Aus Kompatibilitätsgründen kann diese Methode auf öffentliche Felder von Instanzen von Java Klassen zugreifen.

createLocalVariableInstruction(...)
generiert den Zugriff auf lokale Variablen.

Der RValueInstructionCreator schreibt dabei die Instruktionen erst einmal in einen Puffer, damit diese verzögert geschrieben werden können.

5.3.17.1. Primärausdrücke

Die Codegenerierung für die Primärausdrücke geschieht im stage3.Stage3PrimaryVisitor. Diese Ausdrücke beschreiben die

konstanten Ausdrücke:
Diese sind die Literale, die in Kapitel 3 definiert wurden.

- Ist das Literal eine Zahl, so wird die Ldc-Klasse verwendet, um diese in den Bytecode zu schreiben. Die Klasse erwartet bei der Instanzerzeugung einen String. Da Zahlen im abstrakten Syntaxbaum als Strings repräsentiert werden, werden diese bei der Erzeugung übergeben. Dann wird die Instruktion zum statischen Methoden Aufruf $valueOf von Number erzeugt und der InstructionList hinzugefügt. Die Methode erwartet einen String als Parameter und generiert aus diesem einen Number. Der erzeugte Bytecode für eine konstante Zahl sieht wie folgt aus:

  ```
  1: ldc Zahl
  2: invokestatic  Number.$valueOf
  ```

  Die Stringrepräsentation von Zahlen ist deshalb notwendig, da Zahlen in TeaJay größer sein dürfen als die Java-Primitiven long und double.

- Ist das Literal true oder false, dann wird ein Objekt der IConst-Klasse erzeugt und in der InstructionList ergänzt. Die Klasse erwartet ein int bei ihrer Erzeugung. Für true wird 1 und bei false 0 übergeben. Die statische Methode valueOf wird von java/lang/Boolean aufgerufen, die ein boolean erwartet und ein Boolean zurückgibt. Der erzeugte Bytecode sieht wie folgt aus:

  ```
  1: iconst 0 // false
  2: invokestatic  Boolean.valueOf
  ```

- Ist das Literal eine Zeichenkette, wird ein Objekt der Ldc-Klasse mit dem String erzeugt und dem Bytecode hinzugefügt.

- Ist das Literal die null-Referenz, wird ein Objekt von AConstNull erzeugt und dem Bytecode hinzugefügt.

Bei allen konstanten Ausdrücken wird der resultierende statische Typ dem verwalteten Typstack hinzugefügt.

geklammerte Ausdrücke:
Der Ausdruck in der Klammer wird traversiert, die Instruktionen generiert und der InstruktionList hinzugefügt, wie bei jedem anderen Ausdruck. Das heißt, bei der Traversierung bekommen geklammerte Ausdrücke eine neue ExpressionEnvironment und somit auch einen neuen Typstack. Sollte es eine alte ExpressionEnviroment-Instanz geben, so wird sie gesichert. Des Weiteren wird ein neuer Infixoperator-Stack aufgebaut. Nachdem der geklammerte Ausdruck erzeugt ist, wird der resultierende Typ des Ausdrucks dem Typstack in der gesicherten ExpressionEnvironment hinzugefügt und diese wird dann weiter genutzt. Die grammatikalische Regel für geklammerte Ausdrücke befindet sich deshalb bei den Haupt- bzw. Primärausdrücken, da hierdurch der Vorrang der Klammerung erreicht wird.

Zugriff auf lokale und Feldvariablen:
Die Bezeichner, die in einem Primärausdruck stehen, können entweder eine lokale Variable, ein Feld, this oder super adressieren. Während es durch den verwendeten Namen leicht ist this- und super-Zugriffe zu identifizieren, ist die Unterscheidung von Zugriffen auf lokale Variablen, Felder und Typen nicht so einfach, da hier geprüft werden muss, wen der Bezeichner adressiert.

Zunächst wird geprüft, ob der Bezeichner eine lokale Variable adressiert. Die Prüfung wird mit der lokalen Tabelle ermöglicht. Ein lesender Zugriff wird durch ALoad und den Index, der in der lokalen Tabelle für die Variable existiert, realisiert. Der schreibende Zugriff wird durch AStore ermöglicht.

Die Realisierung von Feldzugriffen geschieht durch Abfragen des ClassFiles nach Feldern mit dem Feldnamen. Danach wird geprüft, ob das Feld und die Methode, von denen aus zugegriffen wird, nicht-statisch sind. Dies lässt sich leicht durch das ClassFile erfragen. Ist dies der Fall, so wird der lesende Zugriff durch Laden der echten this-Referenz und dem Erzeugen einer Instanz von GetField, welche dem Bytecode hinzugefügt wird, realisiert. Die GetField-Klasse erwartet eine Beschreibung des Feldes mit Namen und Typ. Diese wird aus dem ClassFile ermittelt und bei der Erzeugung mit angegeben. Der schreibende Zugriff geschieht nach folgendem Muster:

1. Laden der echten this-Referenz (ALoad(0))
2. Erzeugen des Ausdrucks, der geschrieben werden soll
3. PutField wird der InstruktionList hinzugefügt, die Erzeugung geschieht wie bei GetField

Ist das Feld statisch, wird beim lesenden Zugriff keine this-Referenz gebraucht. Es wird eine Instanz von GetStatic erzeugt, die bei der Erzeugung den Namen der Entität erwartet, in der sich das Feld befindet und der Typ des Feldes wird auf dem im Compiler verwalteten Typstack gelegt. Handelt es um einen schreibenden Zugriff, so wird PutStatic verwendet.

Nach this und super folgt immer ein Methodenname, der aufgerufen werden soll oder es handelt sich um den this- bzw. super-Aufruf. Da this und super in TeaJay Felder sind, werden sie auch als solche geladen.

Der super-Aufruf:
Bei super-Aufrufen wird zunächst geprüft, ob diese als erste Anweisung im Konstruktor stattfindet. Ist dies nicht der Fall, handelt es sich um einen fehlerhaften Aufruf von super. Damit dem Nutzer ein super-Aufruf nicht aufgezwungen wird, wird während der Traversierung aller Ausdrücke und Anweisungen, außer dem super-Aufrufs, das super-Feld automatisch initialisiert. Dies funktioniert nur, wenn es einen parameterlosen Konstruktor im Supertyp gibt. Die Stage3Environment führt diese Initialisierung durch die Hilfsmethode writeSuper durch und prüft dabei auch die gerade beschriebenen Bedingungen.

zugriff über den Operator[ ]:
Zuerst wird die Variable links neben den eckigen Klammern lesend ausgewertet, wie oben beschrieben. Dann geschieht der lesende Zugriff über [] nach folgendem Schema:

1. Der Ausdruck in den eckigen Klammern wird ausgewertet und geschrieben, nach dem gleichen Prinzip wie bei den Klammerausdrücken.
2. Es wird geprüft, ob der Typ auf dem Typstack die Methode operator_get besitzt. Des Weiteren wird geprüft, ob die Methode mit dem resultierenden Typ des Ausdrucks in den [] aufgerufen werden kann.
3. Die Methode wird aufgerufen.
4. Der Rückgabetyp des Methodenaufrufs wird auf den Typstack gelegt.

Sollten mehrere eckige Klammern nacheinander stehen, so wird das obige Verfahren genau so oft angewendet, wie es eckige Klammern gibt.

Der schreibende Zugriff ist aufwendiger und sieht z.B. so aus $l[0][1] = 10$. Da der Knoten im Syntaxbaum die Anzahl der eckigen Klammern kennt, wird diese abgefragt. Dann werden, wie im obigen Verfahren, die eckigen Klammern abgearbeitet, dabei allerdings nicht die letzte Klammer. Dort wird der Ausdruck in den [] noch generiert und die resultierenden Typen des Ausdrucks und der letzten *operator_get*-Aufrufs in der ExpressionEnvironment gespeichert. Dann wird die rechte Seite der Zuweisung bearbeitet und generiert. Ist dies geschehen, werden die beiden gemerkten Typen geholt und die Methode operator_set wird aufgerufen.

Methoden- und Konstruktoraufrufe:
Diese werden in Abschnitt 5.3.17.5 behandelt.

5.3.17.2. Arithmetische Ausdrücke

Die arithmetischen Ausdrücke werden in der Grammatik durch die Expression2Rest-Regel beschrieben. Die Regel sieht wie folgt aus:

Expression2Rest:
InfixOperator Expression3

InfixOperator:

```
+ | * | - | / | && | || | < | > | >= | <= | == | !=
```

Das bedeutet allerdings, dass der Operatorvorrang nicht durch den abstrakten Syntaxbaum dargestellt wird und deshalb bei der Traversierung aufgebaut werden muss. Des Weiteren ist es nötig, die arithmetischen Ausdrücke in Postfix Notation darzustellen, da ansonsten der Bytecode nicht generiert werden kann. Beispielsweise wird der Ausdruck $2 + 3 * 4$ durch den Compiler in diesen Bytecode übersetzt:

```
0:  ldc           2
2:  invokestatic  \\Number.$valueOf(Number)
5:  ldc           3
7:  invokestatic  \\Number.$valueOf(Number)
10: ldc           4
12: invokestatic  \\Number.$valueOf(Number)
15: invokestatic  \\Number.mult(Number)
18: invokestatic  \\Number.add(Number)
```

Die Transformation von Infixausdrücken in Postfixausdrücke geschieht mittels der Verwaltung eines Stacks und der Rangfolge der Operatoren. In der Implementierung ist ein Operator vorrangiger als ein anderer, wenn er einen höheren Rang besitzt.

Transformation von Infixausdrücken in Postfixausdrücke:
Das hier beschriebene Verfahren ist von Dijkstra entworfen und in [Dij63] dargestellt.

```
1  operatorstack ← initialisiere Stack
2  Lese Eingabe zeichenweise von links nach rechts
3     schreibe gelesenes Zeichen in Resultat
4     lese nächstes Zeichen
5     falls kein nächstes Zeichen existiert,
6           schreibe alle Operatoren, die auf dem Stack liegen in ↩
               ↪das Resultat
7           beende Schleife
8     falls Operatorstack leer ist,
9           pushe Operator in den Operatorstack
10          und wiederhole die Schleife
11    Prüfe, ob das aktuelle Element des Operatorstacks einen
12    größeren oder gleichen Rang hat als der gelesene Operator
13    falls positiv,
14     schreibe alle Operatoren, die auf dem Stack
15     liegen, in das Resultat, solange das Stackelement einen
16     größeren oder gleichen Rang hat wie der gelesene Operator
17          pushe gelesenen Operator in den Operatorstack
18          und wiederhole die Schleife
19    falls negativ,
20       pushe Operator in den Operatorstack
21 schreibe alle Operatoren, die auf dem Stack liegen, in das ↩
      ↪Resultat
```

Nach diesem Verfahren lässt sich beispielsweise die Eingabe $2+3*4 == 3*4+2$ wie folgt in Postfixnotation transformieren: Das Resultat ist zu Beginn leer und der Stack wird leer initialisiert.

1. Das Resultat = 2.
2. + wird gelesen. Da der Stack leer ist, wird + auf den Stack gelegt.
3. Die Schleife wird wiederholt und das Resultat = 2 3.
4. * wird gelesen. Der Stack ist nicht leer und * hat einen höheren Rang als + . Also wird * auf den Stack gelegt.
5. Die Schleife wird wiederholt und das Resultat = 2 3 4.
6. == wird gelesen. Der Stack ist nicht leer und das oberste Element des Stacks hat einen niedrigeren Rang als *. Der gesamte Stack wird geschrieben, da auch + einen höheren Rang hat als ==. Das Resultat = 2 3 4 * + und == wird auf den Stack gelegt.
7. Die Schleife wird wiederholt und Resultat = 2 3 4 * + 3.
8. * wird gelesen. Der Stack ist nicht leer und * hat einen höheren Rang als == also wird * auf den Stack gelegt.
9. Die Schleife wird wiederholt und das Resultat = 2 3 4 * + 3 4.
10. + wird gelesen. Der Stack ist nicht leer und + hat einen niedrigeren Rang als *. Jetzt wird * in das Resultat geschrieben, aber nicht ==, da == einen niedrigeren Rang als + hat. Das Resultat = 2 3 4 * + 3 4 * und + wird in den Stack gelegt.
11. Die Schleife wird wiederholt und das Resultat = 2 3 4 * + 3 4 * 2 und die Schleife wird beendet.
12. Die verbliebenen Stackelemente werden herausgeschrieben und das Resultat = 2 3 4 * + 3 4 * 2 ==.

Dieses Verfahren ist in der Traversierung von Ausdrücken im abstrakten Syntaxbaum eingebaut. Dabei erzeugt die Implementierung sofort den Bytecode für einen Ausdruck. Allerdings sind alle arithmetischen und Vergleichsoperatoren in TeaJay Methoden. Hinzu kommt, dass es durch die Operatorüberladung nicht möglich ist, einen festen Satz an Methoden anzunehmen, denn sie müssen gesucht werden. Da die Methoden, die einen Infixoperator repräsentieren, alle zwei Parameter erwarten, werden die letzten beiden Typen, die sich auf dem Typstack befinden, betrachtet. Die Methode wird, wie in Kapitel 3 beschrieben gesucht. Anschließend wird der Aufruf in den Bytecode geschrieben. Das Suchen und Schreiben der Operatoren geschieht in der InstructionCreater-Klasse, die sich im Paket codegeneration.instruction befindet.

Die logischen Operatoren && und || können nicht durch eine Methode dargestellt werden, da sie Kurzschlüsse in der Auswertung eines Ausdrucks erlauben. Die Kurzschlüsse werden durch bedingte Anweisungen dargestellt. Zum Beispiel lässt sich der Ausdruck $a < b$ && $b < c$ wie folgt in Java übersetzen:

```
1 a < b ? ( b < c ? : true ) : false
```

Ein solcher Ausdruck lässt sich dann zu diesen Instruktionen übersetzen:

```
1   aload 1 // lade a
2   aload 2 // lade b
3   invokestatic operator_less // ruft Vergleichoperator auf
4   ifeq writeFalse // prüft, Rückgabewert auf false. Wenn ja springe ↩
       ↪zu writeFalse
5     aload 2 // lade b
6     aload 3 // lade c
7     invokestatic operator_less
8     ifeq writeFalse
9  writeTrue:
10       iconst 1 // legt true auf den Stack
11       goto jmpEnd // springe zum nächsten Anweisungsbeginn
12 writeFalse:
13       iconst 0 // legt false auf den Stack
14 jmpEnd:
```

Um während der Traversierung von Ausdrücken mit Kurzschlüssen nicht zu viele Informationen zu verwalten, wird der Aufbau der Anweisung in der ShortCiruitEnvironment, die sich im Paket codegeneration.environment befindet, verwaltet. Anstatt die If-Instruktion in die InstructionsList zu schreiben, werden die Placeholder-Instruktionen geschrieben. Am Ende der Traversierung der Anweisung werden die Placeholder-Instruktionen durch die richtigen If-Instruktionen ausgetauscht. Dabei sind drei Fälle möglich:

1. Fall: Nur logische Unds kommen im Ausdruck vor:
   Alle Placeholder-Instruktion werden durch eine Instanz von If0 mit Prüfung auf Gleichheit und der Label-Instanz zum Ziel writeFalse ersetzt.

2. Fall: Nur logische Oder kommen im Ausdruck vor:
   Alle Placeholder-Instruktionen bis auf die letzten werden durch eine Instanz von If0 mit Prüfung auf Ungleichheit und der Label-Instanz zum Ziel writeTrue. Die letzte Instruktion wird durch If0 mit Prüfung auf Gleichheit ersetzt und der Label-Instanz zum Ziel writeFalse. Sind alle vorherigen Prüfungen zu false ausgewertet worden, dann muss die letzte Prüfung true ergeben, damit der gesamte Ausdruck wahr wird.

3. Fall: Ausdruck besteht aus Kombination von logischen Und und Oder:
   Ausdrücke, die aus einer Kombination von logischen Und und Oder bestehen, können die Sprungziele der If-Instruktionen nicht direkt auf writeFalse oder writeTrue setzen. Zum Beispiel muss der Ausdruck a && $b||c$, die Auswertung von b überspringen wenn a zu false ausgewertet wird, aber c ist noch auszuwerten. Da der Ausdruck in der ShortCircuitEnvironment repräsentiert wird, lässt sich nach diesem Verfahren vorgehen:

   1. Ist der erste Teilausdruck mit einem logisches Und verknüpft, wird der erste Teilausdruck gesucht der rechts neben dem logischen Oder steht.
   2. Ist der erste Teilausdruck mit einem logischen Oder verknüpft, wird der erste Teilausdruck gesucht der rechts neben dem logischen Und steht.
   3. Im ersten Fall werden alle Placeholder-Instruktion-Instanzen bis zum logischen Oder durch Instanzen von If0 mit Prüfung auf Gleichheit und dem

Ziel der ersten Instruktion des Oder-Teilausdrucks ersetzt. Dabei wird die Placeholder-Instruktion, die das Oder repräsentiert, nicht ersetzt. Hierdurch wird gewährleistet, dass die Prüfung für den Ausdruck rechts neben dem Oder durchgeführt wird.

4. Im zweiten Fall werden alle Placeholder-Instruktionsinstanzen bis zum logischen Und durch Instanzen von If0 mit Prüfung auf Ungleichheit und dem Ziel writeTrue ersetzt.
5. Sind alle Teilausdrücke abgearbeitet, wird im ersten Fall die letzte Placeholder-Instruktion durch eine Instanz von If0 mit Prüfung auf Ungleichheit und dem Ziel writeTrue ersetzt. Im zweiten Fall wird die Placeholder-Instruktion durch eine Instanz von If0 mit Prüfung auf Gleichheit und dem Ziel writeFalse ersetzt.
6. Andernfalls wird rekursiv fortgesetzt.

5.3.17.3. Zuweisungen

Zuweisungen können mit den Operatoren =, +=, -=, *= und /= geschehen. Zuweisungen, die mittels =-Operator durchgeführt werden, sind über den LValueInstructionCreator generierbar. Die anderen Operatoren generieren eine Zuweisung über den LValueInstructionCreator, aber zuvor wird die linke Seite des Ausdrucks behandelt, als wäre sie ein R-Value. Das heißt, es wird zuerst der lesende Zugriff auf die Variable in den Bytecode hinzugefügt. Im Anschluss daran wird der Operator, also +, -, * oder /, in den Infixoperator-Stack hinzugefügt und die rechte Seite wird, wie der geklammerte Ausdruck in 5.3.17.1, behandelt. Somit lassen sich die Ausdrücke der Form a*=1+1 behandeln wie a= a*(1+1).

5.3.17.4. Präfixausdrücke

Besitzt ein Ausdruck einen Präfixoperator, so merkt sich die ExpressionEnvironment diesen Operator. Dann wird ein Expression3 Knoten im abstrakten Syntaxbaum traversiert. Ist diese traversiert, wird geprüft, ob sich in der ExpressionEnvironment ein Präfixoperator befindet. Ist dies der Fall, wird der resultierende Typ des Teilausdrucks über den Typstack ermittelt und es wird geprüft, ob dieser die Methode operator_sub hat. Dann wird diese aufgerufen. Ist der Präfixoperator das logische Nicht, wird geprüft, ob der Typ ein Boolean ist, der auf dem Typstack liegt. Es wird folgendes Schema angewendet:

- es wird die Methode booleanValue aufgerufen
- es wird eine 1 auf den Bytecodestack gelegt
- dann wird xor aufgerufen
- es wird das Resultat mit der statischen Methode valueof des Typs Boolean in einen Boolean umgewandelt

Dies führt zu folgendem Bytecode:

```
6:  invokevirtual  Method java/lang/Boolean.booleanValue:()Z
9:  iconst_1
10: ixor
11: invokestatic   Method java/lang/Boolean.valueOf:(Z)Ljava/lang/Boolean;
```

5.3.17.5. Methoden- und Konstruktoraufrufe

MK

Bei Methodenaufrufen ist die größte Herausforderung, den Kontext des Aufrufs zu bestimmen. In TeaJays Grammatik wird nicht zwischen statischen Methodenaufrufen, nicht-statischen Methodenaufrufen und Closure-Aufrufen unterschieden. Das heißt, diese Information muss aus dem Kontext des Aufrufs erschlossen werden. Um diese Problematik zu entschärfen, erlaubt TeaJay keine statischen Methodenaufrufe über Instanzobjekte. Zudem vermischen Closures den Namensraum für Methoden teilweise mit dem Namensraum für Variablen.

Wird ein Methodenaufruf traversiert, so wird zuerst versucht, den Kontext zu bestimmen. Dabei kann nicht immer endgültig entschieden werden, ob es sich z.B. um einen statischen Methodenaufruf oder den Aufruf einer privaten Instanzmethode handelt. Dies ist beispielsweise der Fall, wenn es sich um einen Methodenaufruf ohne vorangestellten Bezeichner handelt:

```
1 ...
2 private static void test(String str) {
3   test(Number.parseNumber("5"));
4 }
5 private void test(Number a) { ... }
6 ...
```

Bevor nicht die Argumente traversiert wurden, kann nicht entschieden werden, welche der beiden Methoden aufgerufen werden soll. Beim Aufruf einer privaten Instanzmethode muss aber, bevor die Argumente ausgewertet werden, die this-Referenz (aload_0) auf den Operanden-Stack gelegt werden. Daher wird in beiden Fällen die this-Referenz auf den Operanden-Stack gelegt und falls sie nicht gebraucht wird, nach dem Methodenaufruf vom Stack entfernt (pop). Hier wird auch der Grund, warum öffentliche Methoden immer über this aufgerufen werden müssen, klar: Es wäre im Vorhinein nicht entscheidbar, ob eine private oder öffentliche Methode aufgerufen werden soll. Daher ist vorab nicht bekannt, ob die tatsächliche this-Referenz (aload_0) oder TeaJays this-Feld auf den Operanden-Stack gelegt werden muss. Dieses Problem könnte mit einer vorausschauenden Übersetzung gelöst werden, wurde für den Prototypcompiler jedoch nicht umgesetzt.

Um zu entscheiden, ob es sich um einen Closure-Aufruf handelt, wird die lokale Variablentabelle und die Feldtabelle konsultiert. Existiert ein Feld oder eine lokale Variable mit einem entsprechenden Namen und ist es bzw. sie vom Typ Closure<...>(...), so wird ein Closure-Aufruf angenommen. In der momentanen Umsetzung verdeckt eine Closure-Variable daher alle Methoden mit demselben Namen:

```
1 private void test(Number a) {
2   Closure() test = closure() { ... };
3   test(a); //Versucht einen Closure-Aufruf
4 }
```

Das liegt ebenfalls an der Tatsache, dass im Vorhinein nicht entschieden wird, von welchem Typ die Argumente sind und daher nicht bekannt ist, ob der Inhalt der Variablen oder eine this-Referenz auf den Operanden-Stack gelegt werden muss.

Wenn klar ist, dass eine Methode (und kein Closure) aufgerufen werden soll, wird mithilfe des MethodFinders (siehe Abschnitt 5.3.12) eine passende Methode gesucht. Bei einem Closure-Aufruf wird immer die Methode Closure.execute(Object...) aufgerufen. Anschließend werden mithilfe von TypeUtils (siehe Abschnitt 5.3.7) die Typparameter überprüft und so endgültig entschieden, ob die Methode bzw. das Closure aufrufbar ist. Sollte es sich bei dem Besitzer der Methode um java.lang.String, java.lang.CharSequence, java.lang.System oder java.io.File handeln, werden einige Methodenaufrufe, wie in Anhang B.2.3 erwähnt, umgeleitet. Die Methode hashCode() wird dabei ungeachtet des Besitzers immer auf TJObjects.hashCode(Object) umgeleitet, welche den Rückgabetyp Number hat.

VarArg-Methoden werden in TeaJay ähnlich implementiert wie in Java und daher erwarten diese ein Java-Array, welches den variablen Anteil der Argumente enthält. Dazu werden, im Falle eines VarArg-Aufrufs die entsprechenden Argumente in ein Java-Array verpackt. Die Information, wie viele Argumente zum variablen Anteil gehören, wird vom MethodFinder mitgeteilt. Da diese immer oben auf dem Operanden-Stack liegen, können sie einfach in einem Array gesammelt werden:

```
1  //Die Argumente wurden von links nach rechts abgearbeitet und liegen auf dem Operanden-Stack
2  JVMInstructionList ins = ...;
3  ins.pushInt(tailLengthForVarArgs); //anzahl variabler Argumente -> MethodFinder
4  ...
5  [ ins.add(new ANewArray(<VarArg-Type>) ] //für Closures z.B. java/lang/Object
6  ...
7  //Holt den nächsten freien Index der lokalen Variablentabelle, erstellt aber keinen Eintrag
8  int arrayIndex = context.getStage3Env().getLocalTable().getNextId();
9  ins.add(new AStore(arrayIndex)); //Speichere das Array für weitere Verwendung
10 for (int i = tailLengthForVarArgs - 1; i >= 0; i--) {
11   ins.add(new ALoad(arrayIndex));
12   ins.add(new Swap());
13   ins.pushInt(i);
14   ins.add(new Swap());
15   ins.add(new AAStore());
16 }
17 ins.add(new ALoad(arrayIndex));
18 //Ein Array mit allen variablen Argumenten liegt nun auf dem Stack
```

Quellcode 5.23: MethodInvokeCreator.java

VarArg-Methoden mittels Java-Arrays umzusetzen, hat den Vorteil, dass die Typinformationen des variablen Parameteranteils in einem Laufzeittyp gespeichert werden können:

```
1  void print(Number... nums); -> void print(Number[] nums);
```

Würde eine VarArg-Methode als letztes Argument eine TeaJay-Liste akzeptieren, so wäre Folgendes nicht möglich:

```
1  public typedef MyType {
2    void print(Object... objs); //-> print(List<Object> objs)
3    void print(Number... nums); //-> print(List<Number> nums)
4  }
```

Durch die Typlöschung hätten beide Methoden dieselbe Laufzeitsignatur:

```
1  void print(List lst);
```

Daher werden VarArg-Methoden wie in Java erstellt und anschließend Code eingefügt, der das Java-Array in eine entsprechende List<...> umwandelt.

Konstruktoraufrufe sind grammatikalisch leichter zu erkennen, denn sie werden immer mit dem Schlüsselwort new eingeleitet. Sie werden in TeaJay grundsätzlich von invokedynamic durchgeführt. Ist der angegebene Typ eine Typdefinition (in TeaJays Sinne), so wird das Laufzeitsystem angewiesen eine Implementierung zu suchen

und den Aufruf auf deren Konstruktor umzuleiten. Da Konstruktoren von Implementierungen immer als erstes Argument eine Referenz auf ihre Identität erwarten, muss invokedynamic eine entsprechende Signatur mitgeteilt werden. Dabei wird auch angenommen, dass der Konstruktor eine Referenz auf das neu erzeugte Objekt zurückgibt. Für den Aufruf

```
1 new List<String>();
```

wird invokedynamic die Signatur

```
1 List <init>(List rthis) //gelöschte Typen, da Laufzeit
```

mitgeteilt. Der tatsächliche Konstruktor hat aber keinen Rückgabewert. Javas Reflection-API unterstützt aber das dynamische Umwandeln von Methodensignaturen (MethodHandle.asType(MethodType)). Die Bootstrap-Methode Bootstraps.bc(...) wandelt daher die Signatur des Konstruktors entsprechend um. Zudem wird über den Eintrag im BootstrapMethods-Attribut der zu erzeugende Typ übergeben (z.B. List). Bei expliziten Konstruktoraufrufen kann als erstes Argument immer null übergeben werden. Konstruktoraufrufe von Implementierungen laufen ähnlich ab, mit der Ausnahme, dass für diese natürlich keine Implementierung gesucht werden muss. Java-Klassen werden ähnlich wie Implementierungen erzeugt, jedoch benötigen diese keine Referenz auf ihre Identität. Die Bootstrap-Methode sucht nur dann nach einer Implementierung, wenn der übergebene Typ abstrakt ist (also den Java-Zugriffsmodifizierer ABSTRACT gesetzt hat) und erzeugt ansonsten den übergebenen Typ direkt. Es obliegt dabei dem Compiler zu prüfen, ob es sich dabei um einen instanziierbaren Typ handelt: Wird z.B. versucht eine abstrakte Java-Klasse zu erzeugen, lehnt der Compiler dies ab. Die Bootstrap-Methode würde aber nach einer Implementierung suchen.

Es ist technisch nicht notwendig, auch Java-Klassen über invokedynamic zu erzeugen, doch erleichtert es die Codegenerierung: Der Konstruktor kann ähnlich einer statischen Methode, welche das Instanzobjekt als Rückgabewert hat, aufgerufen werden. Der Java-Compiler erzeugt Konstruktoraufrufe dagegen wie folgt:

```
1 Aufruf: new Boolean("true");
2 //Bytecode
3 0: new           class java/lang/Boolean
4 3: dup           //Dupliziert das oberste Stackelement
5 4: ldc           String true
6 6: invokespecial java/lang/Boolean."<init>":(Ljava/lang/String;)V
7 //Referenz ist nun initialisiert.
```

Dabei legt die Instruktion new eine uninitialisierte Referenz auf den Operanden-Stack. Das erschwert die Verwaltung der *stack map frames* und des Compiler eigenen Typstacks.

Die Details der Implementierungssuche und die Bootstrap-Methoden werden in Abschnitt 5.2.2.1 bzw. 5.2.2.2 erläutert.

5.3.18. Kompilierung von Statements

MB

Nachdem in Stufe 2 des Kompilierprozesses alle Methodenköpfe der Typimplementierung erzeugt sind, werden in Stufe 3 die Methodenkörper im abstrakten Syntaxbaum traversiert und die Codegenerierung durchgeführt. Beim Besuch einer Methode oder

eines Konstruktors wird der Zustandsautomat in der Stage3Environment Instanz initialisiert. Der Zustandsautomat verwaltet die Zustände, die der Compilier beim Traversieren von Statements erreichen kann. Ein continue-Zustand kann zum Beispiel nur erreicht werden, wenn vorher ein Schleifenzustand existiert. Der Klassenaufbau des Zustandsautomaten ist in Abbildung 5.6 dargestellt. Der Zustandsautomat wird

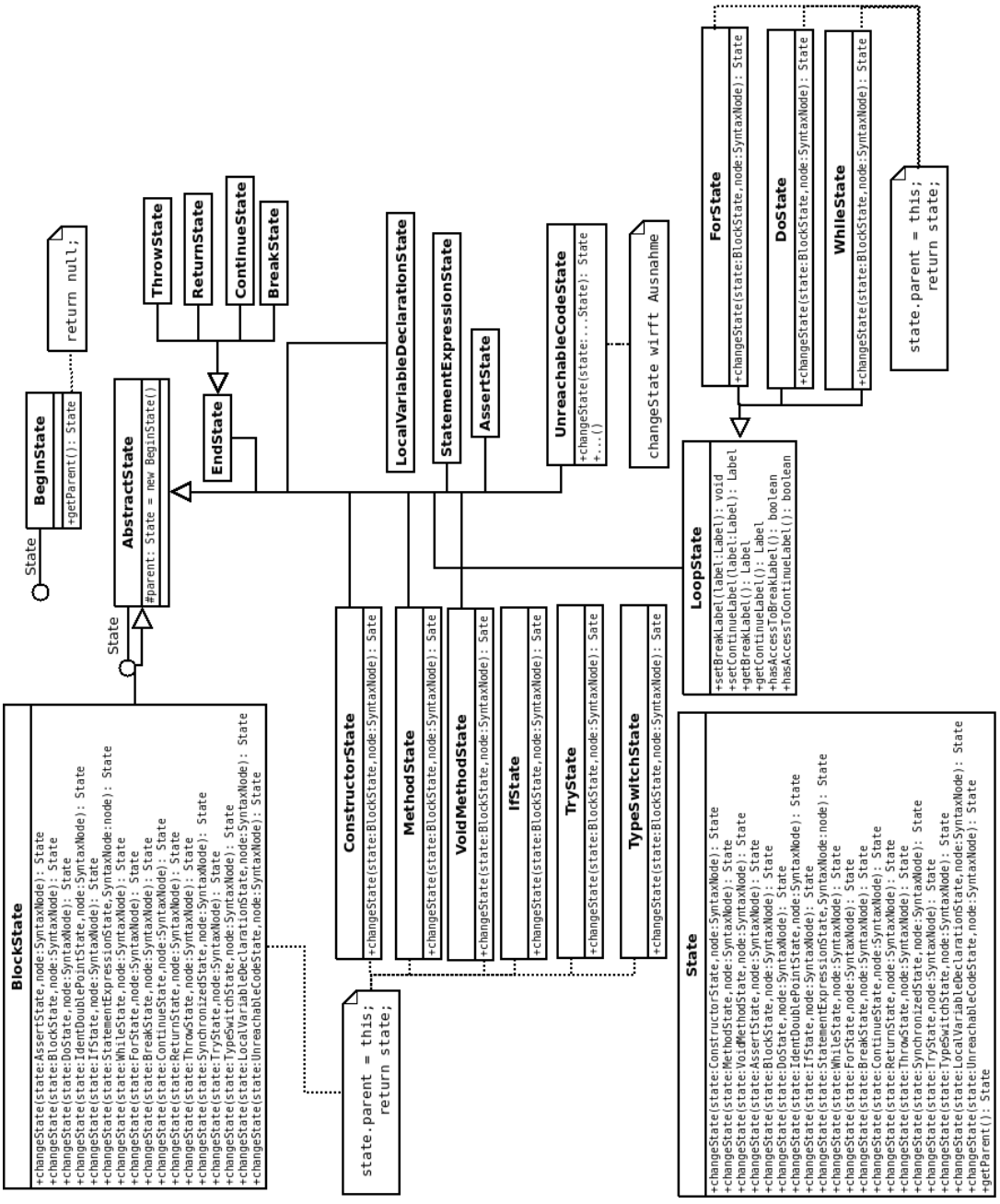

Abbildung 5.6.: Klassenaufbau Zustandsautomat

durch Verkettung von State-Instanzen realisiert. Das bedeutet, es ist möglich, vom aktuellen Zustand in die schon durchlaufenen Zustände zurück zu wechseln. Der Beginn der Verkettung wird durch eine konkrete Instanz von BeginState symbolisiert. Die möglichen Startzustände sind Instanzen von MethodState, ConstructorState und

VoidMethodState. Diese ermöglichen nur einen Wechsel in den Blockzustand. Von den Zustandsklassen TryState, ContinueState, ReturnState und BreakState ist nur der Wechsel in den unerreichbaren Codezustand möglich. Vom BlockState sind alle Zustände bis auf die Startzustände erreichbar.

Des Weiteren werden in einigen Zustandsklassen zusätzliche Informationen gespeichert, die für die Codegenerierung wichtig sind. Die BlockState-Klasse speichert Informationen über verwendete Labels. Die LoopState-Klasse speichert die Labels für den Sprung aus der Schleife und zur Wiederholung der Schleife. Die TryState-Klasse speichert die notwendigen Adressen des Bytecodes, um die ExceptionTable zu füllen.

Die Kommunikation mit dem Zustandsautomaten ist in der Stage3Environment realisiert. Diese bietet die gleiche Schnittstelle zur Kommunikation an, wie das State-Interface, deligiert die Aufrufe aber an den Zustandsautomaten weiter. Zusätzlich implementiert Stage3Environment zwei Hilfsfunktionen die prevState, um in die vorherigen Zustände zurück zu wechseln, und die hasState, um zu prüfen, ob in der Verkettung ein bestimmter Zustand existiert.

Der prinzipielle Aufbau für die Codegenerierung von Anweisungen folgt meistens dem gleichen Muster. Zu Beginn der Traversierung einer Anweisung wird in den Zustand gewechselt, der die Anweisung repräsentiert. Dann werden Ausdrücke und Blöcke generiert. Dabei werden die Ausdrücke geprüft, ob sie die richtigen statischen Typen erzeugt haben. Zum Beispiel bei der if-Anweisung, ob der Ausdruck einen Wahrheitswert ergibt. Am Ende der Traversierung wird in den Zustand vor dem Aufruf zurück gewechselt, falls es möglich ist. Ist am Ende der Traversierung der Zustandsautomat in einem unerreichbaren Codezustand, dann dürfen keine weiteren Anweisungen mehr folgen. Bis zum Beispiel der aktuelle Block abgearbeitet wurde. Dann kann der ursprüngliche Zustand wiederhergestellt werden. Aufgrund des ähnlichen Aufbaus der Anweisungen werden nur einige ausgesuchte Anweisungen beschrieben.

5.3.18.1. typeswitch-Statement

Zu Beginn der Traversierung des typeswitch-Statements wechselt der Zustandsautomat in den typeswitch-Zustand. Danach werden die Fälle des typeswitches traversiert. Zuerst wird geprüft, ob der angegebene Bezeichner eine lokale Variable ist. Diese Prüfung geschieht über die lokale Tabelle. Anschließend werden die angegebenen Falltypen betrachtet. Von jedem Typ wird die ClassFile-Instanz ermittelt und mit dieser lässt sich prüfen, ob der Typ ein Interface, eine Typdefinition, ein Aufzählungstyp oder eine Java-Klasse ist. In allen anderen Fällen wird ein Fehler geworfen. Bei der Traversierung der Fall-Blöcke wird eine Liste von Pair-Instanzen erzeugt. Das Pair hat als erstes Attribut den zu prüfenden Typ und als zweites den Blockknoten des Fall-Blocks. Nachdem alle Fall-Blöcke betrachtet sind, befinden sie sich in der verwalteten Liste aller zu prüfenden Typen mit ihren Fall-Blöcken. Diese Liste wird anhand der Tiefe der Typen im Ableitungsbaum sortiert. Angegebene Interfaces stehen immer zuletzt in der Liste. Sie sind dort ebenso sortiert. Der default-Fall wird daran erkannt, dass im Pair das erste Attribut nicht gesetzt wurde. Dieses Listenelement steht immer am Ende der Liste.

Am Ende der Traversierung der Fall-Blöcke wird die Codegenerierung durchgeführt. Dazu wird die sortierte Liste verwendet. Dabei wird wie folgt vorgegangen:

1. Es wird ein Zähler angelegt, der die Anzahl der Blöcke, die den Endzustand erreichen, verwaltet.

2. Es wird ein Label erzeugt, welches das Ziel des nächsten Fallblocks markiert.

3. Es wird das erste Listenelement betrachtet.

4. Ist das Element der default-Fall, wird der lokalen Tabelle ein neuer Frame hinzugefügt und der Block im Listenelement wird durch den Stage3BlockVisitor traversiert. Dadurch wird der Bytecode für den default-Fall generiert. Im Anschluss wird der Frame wieder aus der lokalen Tabelle entfernt.

5. Falls das erste Element nicht der default-Fall ist, wird die lokale Variable geladen. Danach wird eine Instanz von InstanceOf mit dem Typ, der im Listenelement steht, erzeugt und dem Bytecode hinzugefügt.

6. Es wird dem Bytecode eine Instanz von If0 mit Prüfung auf Gleichheit und dem Label zum nächsten Fall hinzugefügt.

7. Danach wird die lokale Variable geladen und eine Instanz von CheckCast mit dem Typ des Listenelements erzeugt. Hierdurch wird die lokale Variable in den angegebenen Typ umgewandelt.

8. Im Register der lokalen Variablen wird das Resultat von CheckCast geschrieben und der Eintrag der lokalen Variablen in der lokalen Tabelle wird um ein typeswitch-Typ erweitert. Damit Zuweisungen an die Variable zu einem Fehler führen, ist der typeswitch-Typ von der Form ? extends CaseType. Zu diesem Typen ist nur null zuweisbar.

9. Ein Frame wird der lokalen Tabelle hinzugefügt und es wird zur Traversierung eine Instanz von AdaptedTypeswitchBlockVisitor genutzt. Anschließend wird der Frame aus der lokalen Tabelle entfernt.

10. Nach dieser Durchführung wird geprüft, ob der Zustandsautomat in einen Endzustand gewechselt hat. Ist dies der Fall, so wird der Zähler um eins erhöht.

11. Ist kein Endzustand erreicht, wird die lokale Variable geladen. Eine CheckCast-Instruktion mit dem ursprünglichen Typ wird dem Bytecode hinzugefügt und an den Index der lokalen Variablen gespeichert. Hierdurch wird der ursprüngliche Typ wiederhergestellt. Des Weiteren wird der typeswitch-Typ aus der lokalen Tabelle entfernt und ein goto mit einem Ziel der ersten Instruktion außerhalb des typswitches dem Bytecode hinzugefügt.

12. Mit dem Rest der Liste wird analog zum geschilderten Prozess verfahren.

Der Austausch des Besuchers zum Traversieren der nicht-default-Blöcke ist nötig, da bevor dem Bytecode ein return, throw, break oder continue als Anweisung hinzugefügt werden kann, der ursprüngliche Typ wiederhergestellt werden muss. Ansonsten käme es zu einem Problem mit dem Verifizierer der JVM.

Zum Schluss der Traversierung des typeswitch-Statements wird geprüft, ob alle Fall-Blöcke des Statements einen Endzustand erreicht haben. Ist dies der Fall, so wird in den unerreichbaren Zustand gewechselt. Ist dies nicht der Fall, wird das generierte Label, welches die goto-Instruktionen als Ziel nimmt, an den aktuellen Block zur Verwaltung übergeben und NOP wird dem Bytecode hinzugefügt. Somit haben alle goto-Sprünge aus den Fall-Blöcken diese NOP-Instruktion als Ziel. Der Zustandsautomat wird in den Zustand vor der Traversierung versetzt.

5.3.18.2. while- und do-Statement

Zu Beginn der Traversierung wird in den While-Zustand gewechselt. Daraufhin wird eine Label-Instanz erzeugt, die an die erste Instruktion der while-Bedingung gebunden wird. Dies ist das Ziel für die Wiederholung der Schleife. Der Ausdruck wird generiert und es wird anhand des Typstacks geprüft, ob dieser vom Typ Boolean ist. Falls dem so ist, wird eine Instanz von If0 mit Prüfung auf Gleichheit und dem Label für den Sprung über den while-Block generiert. Dieses sowie das Label für die Wiederholung werden der LoopState-Instanz zur Verwaltung hinzugefügt.

Bevor der while-Block besucht wird, wird der lokalen Tabelle ein neuer Frame hinzugefügt. Der Block wird traversiert. Anschließend wird eine Instanz von Goto mit dem Label zur Wiederholung erzeugt und dem Bytecode hinzugefügt, insofern der Zustandsautomat sich nicht im Endzustand befindet. Der Frame wird aus der lokalen Tabelle entfernt und der Blockzustand wird aus dem Zustandsautomaten entfernt.

Beim do-Statement ist die grundlegende Erzeugung ähnlich zum while-Statement, mit dem Unterschied, dass der Block zuerst erzeugt wird und das die If0-Instanz nicht auf Gleichheit prüft, sondern auf Ungleichheit. Jedoch sind Prüfungen durchzuführen, ob der Zustandsautomat in einen unerreichbaren Zustand wechseln muss, nachdem der do-Block traversiert wurde. Hierfür verfügt die LoopState-Klasse über zwei Methoden hasAccessToBreakLabel und hasAccessToContinueLabel, diese ermöglichen es, zu erfragen, ob innerhalb eines Schleifen-Blocks ein continue- oder break-Statement aufgerufen wurde. Das do-Statement lässt den Zustandsautomaten in den unerreichbaren Zustand wechseln, wenn nach der Traversierung des do-Blocks der Zustandsautomat im Endzustand ist und kein continue- oder break-Statement aufgerufen wurde. Dabei sind folgende Fälle nach der Traversierung des Blocks möglich:

- Nach der Traversierung befindet sich der Zustandsautomat im Endzustand und es wird kein break- bzw. continue-Statement traversiert. Dann ist der while-Ausdruck nicht mehr traversierbar und es kommt zu einen Kompilierfehler, da der Ausdruck nicht mehr erreichbar ist.

- Der Zustandsautomat befindet sich im Endzustand, es wurde aber ein break-Statement traversiert. So wird die Codeerzeugung, also die Traversierung, für den while-Ausdruck unterdrückt, da dieser nicht erreichbar ist.

- Es wurde ein continue-Statement traversiert und der while-Ausdruck wird erzeugt, da dieser erreichbar ist.

5.3.18.3. break-, continue-,throw-Statement

In den break- und continue-Zustand kann der Zustandsautomat nur gelangen, wenn vorher ein while-, do- bzw. for-Zustand erreicht wurde. Ansonsten wird eine Ausnahme geworfen. Break- und continue-Statements werden mit goto realisiert. Bei der Traversierung eines break- oder continue-Statements wird geprüft, ob sich der Zustandsautomat in einem Schleifenzustand befindet. Bei einem break-Statement wird das Label für das Anweisungsende der Schleife aus der LoopState-Instanz genommen und InstructionList eine Goto-Instanz mit diesem Label der hinzugefügt. Analog gilt dies für das continue-Statement.

Bei einem throw-Statement wird zuerst in den throw-Zustand gewechselt. Anschließend wird der Ausdruck traversiert und Code für diesen erzeugt. Es wird überprüft, ob der Ausdruck als Resultat einen zuweisbaren Typ auf Throwable erzeugt. Zum Schluss wird eine Instanz von AThrow erzeugt und dem Bytecode hinzugefügt.

5.4. Exkurs: TeaJay erweitern

In diesem Exkurs wird gezeigt wie TeaJay um eine weitere Anweisung, der repeat-until, erweitert werden kann. Die repeat-until-Anweisung ist eine Schleife die ihren Schleifenkörper solange wiederholt, bis eine Bedingung erfüllt ist. Die restliche Semantik ist analog zur do-Schleife definiert. Die Grammatik für eine solche Schleife lässt sich wie folgt definieren:

repeat-until:

repeat Block *until* ParExpression *;*

Eine Erweiterung der Grammatik von TeaJay erfordert eine Vielzahl von Änderungen bzw. Erweiterungen des MiniCompiler-Projekts. Zuerst ist festzuhalten, dass repeat und until keine Schlüsselwörter der Sprache sind und somit nicht von der lexikalischen Analyse als Schlüsselwörter behandelt werden. Die lexikalische Analyse ist zuerst um diese zwei Schlüsselwörter zu erweitern. Da jedes nicht-Terminal der Sprache eine eindeutige ID besitzt, werden in der Klasse lexer.Sym zuerst zwei IDs für repeat und until vergeben. Die Erweiterung in der Klasse sieht wie folgt aus:

```
1 public class Sym {
2   ...
3   public static final int REPEAT = ++cnt;
4   public static final int UNTIL = ++cnt;
5 }
```

Damit die lexikalische Analyse repeat und until als Teriminale erkennt, wird die Lexer.flex-Datei die sich im Paket lexer befindet, wie folgt erweitert:

```
1   ...
2     "throws"        { return symbol(Sym.THROWS); }
3   /* begin TeaJay keywords */
4     "repeat"        { return symbol(Sym.REPEAT); }
5   "until"         { return symbol(Sym.UNTIL);  }
```

```
6   ...
```

Nach diesen zwei Erweiterungen werden repeat und until als Terminale behandelt und sind somit reservierte Schlüsselwörter, die nicht mehr in TeaJay als Bezeichner genutzt werden können. Der Parser muss als nächstes mit der obigen Grammatik für repeat-until erweitert werden. Dies wird, wie in Kapitel 5.3.3, geschildert durchgeführt und es wird folgende Methode in parser.Parser hinzugefügt:

```
1  public class Parser {
2  ...
3  private RepeatUntilNode parseRepeatUntil()throws CompilerException
4  {
5    Symbol s = lexSymbol;
6    if(accept(Sym.REPEAT))
7    {
8      BlockNode block = this.parseBlock();
9      if(block != null)
10     {
11       ParExpressionNode pExp = this.parseParExpression();
12       if(pExp != null){
13         return new RepeatUntilNode(s,block,pExp);
14       }
15       throw new CompilerExceptioin("...",s);
16     }
17     throw new CompilerExceptioin("...",s);
18   }
19   return null;
20 }
21 }
```

Nachdem die Methode implementiert ist, muss die RepeatUntilNode-Klasse erzeugt und dem Paket abstractsyntaxtree hinzugefügt werden. Die Klasse sieht wie folgt aus:

```
1  public class RepeatUntilNode extends SyntaxNode {
2    private BlockNode block;
3    private ParExpressionNode pExp;
4    public RepeatUntilNode(Symbol s, BlockNode b,ParExpressionNode p){
5      super(s);
6      this.block = b;
7      this.pExp = p;
8    }
9    @Override
10   public void accept(Visitor visitor) throws CompilerException {
11     visitor.begin(this);
12     Visitor v = visitor.visit(this);
13     checkNull(this.block, v);
14     checkNull(this.pExp, v);
15     visitor.end(this);
16   }
17 }
```

Da die Methode zum Parsen der repeat-until-Regel und die Klasse zur Repräsentation der Regel im abstrakten Syntaxbaum existieren, ist es notwendig, dass der Parser auch die Regel erreicht. Da es sich bei der Regel, um eine Anweisung handelt, wird

die parseStatement-Methode der Parserklasse erweitert. Somit sieht die Methode wie folgt aus:

```
1  private StatementNode parseStatement()throws CompilerException
2  {
3    ...
4    RepeatUntilNode ru = this.parseRepeatUntil();
5    if (ru != null) {
6      return new StatementNode(s, ru);
7    }
8    return null;
9  }
```

Die Klasse StatementNode muss erweitert werden, damit dieser die RepeatUntilNode verwaltet. Es ist ein neuer Konstruktur und ein weiteres Attribut hinzuzufügen und die accept-Methode ist zu erweitern, damit die RepeatUntilNode mit traversiert werden kann. Die Klasse sieht wie folgt aus:

```
1  public class StatementNode extends SyntaxNode {
2    ...
3    private RepeatUntilNode ru;
4    public StatementNode(Symbol s, RepeatUntilNode r){
5      super(s);
6      this.ru = r;
7    }
8    @Override
9      public void accept(Visitor visitor) throws CompilerException {
10     ...
11     checkNull(this.ru, v);
12     visitor.end(this);
13   }
14 }
```

Jetzt kann der Parser die repeat-until-Anweisung in einen Knoten des abstrakten Syntaxbaum transformieren. Allerdings ist es noch nicht möglich, diese zu traversieren, da hierfür neue Besucherklassen erzeugt werden müssen und die codegeneration.Visitor-Klasse um die Methoden begin, visit und end zu erweitern ist. Zuerst wird die Klasse erweitert. Dies führt zu folgendem Quelltext, der in die Klasse hinzugefügt wird:

```
1  public abstract class Visitor  {
2    ...
3    public void begin(RepeatUntilNode n)throws CompilerException{
4    }
5    public Visitor visit(RepeatUntilNode n)throws CompilerException{
6      return this;
7    }
8    public void end(RepeatUntilNode n)throws CompilerException{
9    }
10 }
```

Es sind nach dieser Änderung eine abstrakte Klasse AbstractRepeatUntilVisitor im Paket codegeneration.abstractvisitors zu erstellen, damit diese die Methoden vorgibt, um die Kindknoten zu traversieren. Die Klasse sieht wie folgt aus:

```
1 public abstract class AbstractRepeatUntilVisitor extends Visitor {
2   public abstract void begin(BlockNode n)throws CompilerException;
3   public abstract Visitor visit(BlockNode n)throws CompilerException;
4   public abstract void end(BlockNode n)throws CompilerException;
5
6   public abstract void begin(ParExpressionNode n)throws ↩
      ↪CompilerException;
7   public abstract Visitor visit(ParExpressionNode n)throws ↩
      ↪CompilerException;
8   public abstract void end(ParExpressionNode n)throws ↩
      ↪CompilerException;
9 }
```

Da die StatementNode-Klasse einen weiteren Kindknoten verwaltet, ist der AbstractStatementVisitor um die Methode begin, visit und end zu erweitern. Die Klasse sieht wie folgt aus:

```
1 public abstract class AbstractStatementVisitor extends Visitor {
2   ...
3   public abstract void begin(RepeatUntilNode n)throws ↩
      ↪CompilerException;
4   public abstract Visitor visit(RepeatUntilNode n)throws ↩
      ↪CompilerException;
5   public abstract void end(RepeatUntilNode n)throws CompilerException↩
      ↪;
6 }
```

Es ist jetzt noch nicht möglich die repeat-until-Schleife in TeaJay-Quelltext zu schreiben, da für sie noch keine Codegenerierung existiert, weil es keinen konkreten Besucher für die Schleife gibt und der stage3.StatementVisitor diese noch nicht betrachtet. Der Besucher für die Schleife wird im nachfolgenden Quelltext präsentiert:

```
1  public  class Stage3RepeatUntilVisitor extends ↩
     ↪AbstractRepeatUntilVisitor {
2   Stage3Environment stage3Env;
3   JVMInstruction nop = new NOP();
4   BlockVisitor blockVisitor;
5   boolean isEndState = false;
6   public StageRepeatUntilVisitor(Stage3Environment s){
7     this.stage3env = s;
8   }
9   public void begin(BlockNode n)throws CompilerException{
10    BlockState state = new BlockState();
11    stage3Env.getLocalTable().addFrame();
12    stage3Env.changeState(state, node);
13    this.breakLabel = new Label("repeat break label");
14    this.continueLabel = new Label("repeat continue label");
15
16    LoopState loopState = stage3Env.hasState(LoopState.class);
17    loopState.setBreakLabel(this.breakLabel);
18    loopState.setLoopLabel(this.continueLabel);
19    try{
20      stage3Env.getLocalTable().addStackMapFrame(
21
22      stage3Env.getExpressionEnvironment().getMethod().
```

```
23            getCode().getInstructionList().add(nop));
24        }catch(ByteCodeException ex){
25          ...
26        }
27      }
28      public Visitor visit(BlockNode n)throws CompilerException{
29        return blockVisitor = new Stage3BlockVisitor(stage3env);
30      }
31      ...
32    }
```

Zu Beginn der Traversierung des Blocks wechselt der Zustandsautomat in den Blockzustand. Es wird eine NOP-Instruktion der InstructionList der aktuell bearbeiteten Methode hinzugefügt. Die NOP-Instruktion dient der Markierung für den Beginn des repeat-Blocks. Da dieser der zukünftige Sprungziel für die Wiederholung des repeat-Blocks ist, wird an seiner Adresse ein stack map frame gebunden. Des Weiteren werden zwei Label-Instanzen erzeugt, die als Markierer für den Sprung aus der Schleife und für die Wiederholung der Schleife dienen. Das Label für die Wiederholung der Schleife wird an der ersten Instruktion der until-Anweisung gebunden, da die Schleifenbedingung vor der Schleifenwiederholung geprüft werden muss.

```
1   public class Stage3RepeatUntilVisitor {
2     ...
3     public void end(BlockNode n)throws CompilerException{
4       if (blockVisitor.isEmptyBlock()) {
5         stage3Env.getExpressionEnvironment().insertCode(new NOP());
6       }
7       EndState endState = stage3Env.hasState(EndState.class);
8       if (endState != null) {
9         this.isEndState = true;
10      } else {
11        stage3Env.getLocalTable().removeFrame();
12        stage3Env.prevState();
13      }
14    }
15    ...
```

Am Ende der Traversierung eines Blocks wird geprüft, ob der Block eine Anweisung enthält. Ist dies nicht der Fall wird eine NOP-Instruktion in die InstructionList hinzugefügt[5]. Dann wird der Zustandsautomat geprüft, ob er sich im Endzustand befindet. Ist dies wiederum nicht der Fall, wird aus der lokalen Tabelle das hinzugefügte Frame entfernt und der Blockzustand wird aus der Verkettung entfernt. Ansonsten speichert die Klasse, dass sich der Blockzustand in einem Endzustand befindet.

```
1   public class Stage3RepeatUntilVisitor {
2     ...
3       public void begin(ParExpressionNode node) throws ↩
          ↪CompilerException {
4       LoopState lState = stage3Env.hasState(LoopState.class);
```

[5]Dies hat nur eine Bedeutung für die Strukturierung des Bytecodes, damit dieser besser gelesen werden kann

```
5     if (!isEndState || lState.hasAccessToContinueLabel() || lState.↩
          ↪hasAccessToBreakLabel()) {
6       if (isEndState) {
7         stage3Env.prevState(); // forget endstate
8         stage3Env.prevState(); // forget repeat block
9         stage3Env.getLocalTable().removeFrame();
10      }
11      if(lState.hasAccessToContinueLabel() || !isEndState){
12        ExpressionEnvironment env = stage3Env.↩
              ↪getExpressionEnvironment();
13        BlockState state = stage3Env.hasState(BlockState.class);
14        state.addLabel(continueLabel);
15        env.insertCode(new NOP());
16      }
17    } else {
18      throw new CompilerException("Unreachable state ...", node.↩
            ↪getSymbol());
19    }
20  }
21  public Visitor visit(ParExpressionNode n)throws CompilerException{
22    LoopState lState = stage3Env.hasState(LoopState.class);
23    //Unterdrücke codegenerierung für until,
24    //wenn der repeat Block im End Zustand ist
25    //und es kein Continue Statement gibt
26    if (this.isEndState && !lState.hasAccessToContinueLabel()) {
27      return null;
28    }
29    return new Stage3ParExpressionVisitor(stage3Env);
30  }
31  ...
32 }
```

Am Anfang der Traversierung der until-Ausdrucks ist zu prüfen, ob der Block einen Endzustand erreichte und er continue- oder break-Anweisungen enthielt. Ist dies so, sind die Anweisungen außerhalb des repeat-Blocks erreichbar. Deshalb wechselt der Zustandsautomat zurück in den Zustand vor der Traversierung der repeat-Anweisung. Ansonsten wird eine Ausnahme geworfen, da die nachfolgenden Anweisungen nicht erreichbar sind. Enthält der repeat-Block eine continue-Anweisung oder der Zustandsautomat befindet sich nicht im Endzustand, so wird der Instruction-List eine NOP-Instanz hinzugefügt. Diese Instruktion dient der Markierung für den Beginn von until. Durch die Übergabe des continue-Labels an den aktuellen Blockzustand wird das Label an die NOP-Instruktion gebunden. Bei der Erzeugung von until durch den Besucher wird geprüft, ob der Zustandsautomat sich in einen Endzustand befindet und kein continue-Ausdruck im repeat-Block existiert. Ist dies der Fall, so wird die Traversierung des Ausdrucks unterdrückt.

```
1  public class Stage3RepeatUntilVisitor  {
2    ...
3    public void end(ParExpressionNode n)throws CompilerException{
4      LoopState lState = st3Env.hasState(LoopState.class);
5      if (!isEndState || lState.hasAccessToContinueLabel() || lState.↩
           ↪hasAccessToBreakLabel()) {
6        ExpressionEnvironment env = st3Env.getExpressionEnvironment();
7
```

```
8         Type.ClassType bType = new Type.ClassType(null, "java/lang/↩
              ↪Boolean", null);
9         Type typeOfResult = env.typeOfResult();
10        try {
11          st3Env.getTypeUtils().checkAssignability(bType, typeOfResult)↩
                ↪;
12          MethodRefInfo ref = MethodRefInfo.make("java/lang/Boolean", "↩
                ↪booleanValue", "boolean");
13          env.getMethod().getCode().getInstructionList().add(new ↩
                ↪InvokeVirtual(ref));
14          env.getMethod().getCode().getInstructionList().add(new If0(↩
                ↪IMode.eq, nop));
15        }catch (ByteCodeException ex){
16          ex.printStackTrace(System.err);
17          System.exit(-1);
18        }
19        catch (TypeException ex) {
20          throw new CompilerException("do statetement expects Boolean",↩
                ↪ node.getSymbol());
21        }
22        BlockState state = stage3Env.hasState(BlockState.class);
23        state.addLabel(breakLabel);
24        env.insertCode(new NOP());
25    }
26  }
```

Zum Schluss der Traversierung von until wird der Bytecode hinzugefügt, der die Prüfung des Wahrheitswerts auf dem Laufzeitstack durchführt. Ist der Wert dabei false, wird auf die Position der NOP-Instruktion verwiesen, welche die Schleifenwiederholung repräsentiert. Am Ende der Methode wird durch das Einfügen einer weiteren NOP-Instruktion das Anweisungsende markiert. An diese Instruktion ist das break-Label gebunden.

Die Klasse zur Traversierung der Anweisung ist komplettiert, allerdings wird sie im Traversierungsprozess noch nicht verwendet. Dies muss noch in der Stage3StatementVisitor geändert werden. Hierfür werden die Methoden begin,visit und end mit dem Parameter RepeatUntilNode hinzugefügt. Sie sieht dann wie folgt aus:

```
1   public class Stage3StatementVisitor  {
2     ...
3     public void begin(RepeatUntilNode n)throws CompilerException{
4       stage3Env.changeState(new RepeatUntilNode(),s);
5     }
6     public Visitor visit(RepeatUntilNode n)throws CompilerException{
7       return new Stage3RepeatUntilVisitor(stage3Env);
8     }
9     public void end(RepeatUntilNode n)throws CompilerException{
10      stage3Env.prevState();
11    }
12  }
```

Es fehlt noch die Anpassung des Zustandsautomats im Paket stage3.states. Hierfür muss dem State-Interface die Methode State changeState(RepeatUntilState state, SyntaxNode n) hinzugefügt werden. Die AbstractState-Klasse und die BeginState-Klasse müssen diese Methode implementieren, so dass ein Versuch des Zustands-

wechsel eine Ausnahme erzeugt. Die BlockState-Klasse muss die Methode ebenfalls realisieren, allerdings so, dass der Zustandswechsel in die RepeatUnitState geschieht. Es ist noch eine neue Klasse in diesem Paket zu erstellen, der den Zustand für die repeat-until-Anweisung repräsentiert. Die Klasse sieht dann wie folgt aus:

```
1  public class RepeatUntilState extends LoopState {
2      public RepeatUntilState(){
3
4      }
5
6    @Override
7      public State changeState(BlockState state,SyntaxNode node)throws ↩
           ↪CompilerException{
8          state.parent = this;
9          return state;
10     }
11 }
```

Denn vom Zustand der RepeatUntil-State ist es nur möglich, in den Blockzustand zu gelangen. Durch diese hier präsentierten Änderungen wurde der TeaJay-Compiler um einen neuen Ausdruck erweitert.

6. Ausblick

MB
MK

Während der Entwurfsphase von TeaJay diskutierten die Entwickler oft über *Design by contract* und wie die Sprache helfen kann, Verträge zu formulieren und durchzusetzen. Dabei waren die diskutierten Ansätze nicht im Rahmen dieser Arbeit umsetzbar bzw. die Frage nach der generellen Umsetzbarkeit konnte nicht abschließend beantwortet werden. An dieser Stelle werden zwei der diskutierten Ansätze kurz vorgestellt:

Formale Spezifikation von Verträgen :
Verträge sollen in einer speziellen Sprache formulierbar sein und vom Compiler so gut es geht geprüft werden. Dabei soll es zum einen möglich sein, Vor- und Nachbedingungen von Methoden zu spezifizieren und zum anderen auch Invarianten z.B. von Schleifen und Klassen zu formulieren. Ein recht fortgeschrittener formaler Theorembeweiser, der Ausgangspunkt eines solchen Unterfangens sein könnte, ist z.B. Isabelle (siehe [PNW14]).

Testtreiber als Teil der Spezifikation :
Verträge werden informal spezifiziert und als Teil der Spezifikation wird TeaJay-Code angegeben, der die Einhaltung der Verträge zur Laufzeit prüft. Dieses Vorgehen ähnelt dem Benutzen des assert-Statements zur Überprüfung von Vor- und Nachbedingungen. assert darf jedoch nur in Implementierungen vorkommen und legt die Interpretation der Verträge in die Hand des Entwicklers der Implementierung. Dieses Vorgehen hat auch einen großen Nachteil: Der Testtreiber könnte eventuell die Semantik einer Methode beeinflussen und unerwünschte Nebenwirkungen haben. Die Programmiersprache Eiffel (siehe [Eif13]) unterstützt ein ähnliches Konzept.

Während der Entwurfsphase wurde auch diskutiert, parametrisierte Typen in TeaJay reifizierbar (englisch: *reifiable* [GJS+12, Paragraph 4.6]) zu implementieren. Dazu müsste TeaJays Laufzeitsystem auch die Verwaltung von Laufzeittypen übernehmen. Dies erledigt momentan noch die JVM. Reifizierbare, parametrisierte Typen hätten es auch erlaubt, Implementierungen speziell für konkrete Typargumente eines generischen Typs umzusetzen. Das heißt z.B. es wird möglich eine Implementierung von Stack<T> speziell für den Typ Number zu erstellen, welche immer dann geladen wird, wenn ein Stack<Number> angefordert wird. Zudem wäre es mit reifizierbaren Typen möglich, typeswitch deutlich mächtiger zu implementieren. Dadurch würde der unsichere cast-Operator überflüssig.

TeaJay unterstützt interfaces, um das Fehlen von Mehrfachvererbung zu kompensieren. Dabei ist der Unterschied zwischen einem interface und einer Typdefinition nur gering. Bei einer weiteren Entwicklung von TeaJay könnte man diese zusammenlegen und zwei Erbmodi für Typdefinitionen einführen: Das Erben mit bzw. ohne

Implementierung. Dabei dürfte die Implementierung nur von einem Typ, jedoch von den restlichen Typen nur die Schnittstelle geerbt werden.

Ebenfalls wurde diskutiert, ob es sinnvoll wäre, den Zugriffsmodifizierer const für Methoden einzuführen. Er sollte dabei bedeuten, dass die Methode den internen Zustand ihres Objekts nicht verändert. Die technischen Umsetzungsmöglichkeiten waren jedoch alle nicht befriedigend, da entweder const-Methoden zu stark in ihren Möglichkeiten eingeschränkt werden mussten oder der const-Vertrag nicht durch den Compiler durchgesetzt werden konnte. Dabei sei angemerkt, dass selbst der g++ nicht in der Lage ist, const vollständig durchzusetzen. Es besteht dadurch die Gefahr, den Entwickler in falscher Sicherheit zu wiegen. Bei einer Weiterentwicklung von TeaJay wäre es aber sinnvoll, den Zugriffsmodifizierer final, nach dem Vorbild von Java, für Variablen zu erlauben.

Durch das Erlauben von Methodenzeigern könnte TeaJays Anspruch, auch funktionale Programmierung zu unterstützten, verstärkt werden. Dazu müsste jedoch eine neue Syntax eingeführt werden, was die Komplexität der Sprache erhöhen würde. Da momentan bereits Closures als Methodenzeiger dienen können, sollte dieser Schritt noch weiter diskutiert werden.

Es wurde auch eine Erweiterung von TeaJays generischem Typsystem um *union types*, nach dem Vorbild von Ceylon (vgl. [cey14]), diskutiert. Diese sind vermutlich für Anfänger einfacher zu verstehen als TeaJays Wildcard-Typen. Zudem würde TeaJays generisches Typsystem von Typinferenz (englisch: type inference, vgl. [GJS+12, S. 466ff]) profitieren und dessen Benutzung erleichtern. Dies wird vor allem in Abschnitt 4.1.3.3 d) deutlich.

Zudem sollte der Prototypcompiler weiterentwickelt werden, um TeaJay attraktiver zu gestalten. Folgende Eigenschaften sind dabei besonders zu nennen:

- typeswitch auch auf Feldern erlauben
- generische Instanzmethoden
- Java-Kompatibilität verbessern
- überprüfte Ausnahmen
- multithreading Unterstützung (synchronized)
- mehrere Interfacegrenzen in generischen Typsignaturen, wie es z.B. in Java erlaubt ist:

```
1 class MyClass<T extends Object & Iterable<MyClass> & Comparable<MyClass>> {... }
```

7. Fazit

MB
MK

TeaJay gibt einen Einblick, wie eine moderne ADT-orientierte Programmiersprache aussehen kann. Beim Entwurf von TeaJay stand der Lehraspekt an erster Stelle und so ist TeaJay eine Sprache geworden, die stark von der Maschine abstrahiert. Das ermöglicht dem Anwender, sich mit dem zu lösenden Problem zu beschäftigen und nicht mit Eigenarten der Zielmaschine. Hierbei ist vor allem das Konzept von Number als einziger Zahltyp, List und der Verzicht auf primitive Datentypen zu nennen. Die Entscheidung für ein solch hohes Abstraktionsniveau birgt aber auch hohe Laufzeitkosten, welche jedoch auf lange Sicht, durch die Weiterentwicklung und Optimierung von TeaJay verringert werden können. Laufzeitkosten sind für eine Lehrsprache jedoch von nachrangiger Bedeutung. Die Entscheidung TeaJay zur JVM zu übersetzen, hat zwar Einfluss auf einige Entwurfsentscheidungen genommen, jedoch bietet die JVM den enormen Vorteil der Plattformunabhängigkeit. Es ist daher möglich, TeaJay auf beinahe allen weitverbreiteten Systemen einzusetzen. Dies ist eine konsequente Fortführung des hohen Abstraktionsniveaus von TeaJay. TeaJay ist zudem eine Sprache, die in ihren Fähigkeiten mit aktuellen Programmiersprachen vergleichbar ist. Dies wird vor allem in Kapitel 4 deutlich. TeaJay kann auf viele Java-Bibliotheken zurückgreifen und daher leicht um eigene Bibliotheken erweitert werden. Sie bietet außerdem moderne Sprachmittel wie z.B. typeswitch, generische Typen und generische Methoden. typeswitch vereinfacht den sicheren Umgang mit Laufzeittypen und beeinflusst den Sprachnutzer ebenfalls im Denken: Die Angabe des default-Falls ist nicht optional und soll den Sprachnutzer dazu bringen, sich mit dem Verhalten für diesen Fall gedanklich zu beschäftigen. Generische Typen bzw. Methoden erweitern die Entwurfsmöglichkeiten für Schnittstellen. Der Prototypcompiler für TeaJay setzt die meisten Sprachmittel umfassend um und kann dazu dienen, die Akzeptanz von TeaJay zu untersuchen.

TeaJays Hauptentwurfsziel war es, das Formulieren von ADTs zu ermöglichen. Dabei sollte die Spezifikation eines Typs so stark wie möglich von der Implementierung getrennt werden. Dieses Ziel wurde erreicht, indem eine Laufzeitumgebung die Implementierung eines Typs erst zur Laufzeit sucht und bindet. Ein Entwickler kann sich daher zur Kompilierzeit nur auf die Informationen, welche ihm die ADT-Spezifikation bietet, verlassen und weiß nicht, mit welcher Implementierung er kommunizieren wird. TeaJay beeinflusst dadurch das Denken des Sprachnutzers und forciert die gedankliche Unterscheidung zwischen Spezifikation und Implementierung. Der Typ Closure ist ein weiteres Sprachmittel, welches das Denken des Sprachnutzers beeinflusst. Durch Closures wirkt es natürlicher als in Java (Version 7), funktionale Programme zu formulieren und der Sprachnutzer wird dadurch angeregt, sich mit funktionalen Konzepten auseinanderzusetzen.

Ein weiteres Ziel des Entwurfs von TeaJay war, dass eingebaute und benutzerdefinierte Typen sich möglichst gleichberechtigt in der Sprache darstellen. Dieses Ziel

wurde vor allem durch das Erlauben von Operatorüberladung und den Verzicht auf primitive Typen sowie Arrays (im Sinne von Java) erreicht. Hinzu kommt, dass alle Typen in TeaJay Referenztypen sind. Aus technischen und pragmatischen Gründen haben die eingebauten Typen aber noch immer eine gewisse Sonderstellung. Diese tritt jedoch nicht mehr so stark in den Vordergrund, wie z.B. bei Java.

Die Entwicklung einer Programmiersprache ist ein dynamischer Prozess und ihre Bewertung hängt von subjektiven Kriterien ab. Daher ist abschließend zu sagen, dass TeaJay die Ziele, die sich die Autoren zu Beginn der Arbeit gesetzt haben, erfüllt.

A. Die TeaJay-Grammatik

Die hier angegebene Grammatik leitet sich von der in [GJS+12, S. 591ff] angegebenen Java-Grammatik ab. Unterschiede zwischen der Java Grammatik und der TeaJay Grammatik werden durch ein (*) gekennzeichnet. Regeln die geparst werden, aber in TeaJay nicht verwendet werden und vom Compiler mit einer Fehlermeldung behandelt werden, sind mit (†) gekennzeichnet.

QualifiedIdentifier:

Identifier { . Identifier }

QualifiedIdentifierList:

QualifiedIdentifier { , QualifiedIdentifier }

CompilationUnit(*):

[CompilationUnitHelper] { ImportDeclaration } (TypeDef | TypeDeclaration)

CompilationUnitHelper:

[Annotations] "package" QualifiedIdentfier ";"

ImportDeclaration:

"import" ["static"] QualifiedIdentfier [".*"] ";"

TypeDeclaration(*):

ClassOrInterfaceDeclaration | TjImplDeclaration | ";"

TypeDef(*):

[Modifier] "typedef" Identifier [TypeParameters] ["extends" Type]
["interfaces" TypeList] TjTypeDefBody

TjTypeDefBody(*):

" { TjTypeDefBodyDeclaration } "" "

TjTypeDefBodyDeclaration(*):

[";"] | { Modifier } TjTypeDefMemberDecl

TjTypeDefMemberDecl(*):

TjTypeDefConstructorDecl | Type Identifier InterfaceMethodDeclaratorRest |
"void" Identifier VoidInterfaceDeclaratorRest | InterfaceGenericMethodDecl

TjTypeDefConstructorDecl(*):

Identifier InterfaceMethodDeclaratorRest

TjImplDeclaration(*):

"typeimpl" Identifier [TypeParameters] "implements" Type ClassBody

ClassOrInterfaceDeclaration:

{ Modifier } (ClassDeclaration | InterfaceDeclaration)

ClassDeclaration:

NormalClassDeclaration | EnumDeclaration

InterfaceDeclaration:

NormalInterfaceDeclaration | AnnotationDeclaration

NormalClassDeclaration(†):

"class" Identifier [TypeParameters] [extends Type]
["implements" TypeList] ClassBody

EnumDeclaration:

"enum" Identifier ["implements" TypeList] (†) EnumBody

NormalInterfaceDeclaration:

"interface" Identifier [TypeParameters] ["extends" TypeList] InterfaceBody

AnnotationTypeDeclaration(†):

"@" "interface" Identifier AnnotationTypeBody

Type(*):

BasicType { "[" "]" } (†) | ReferenceType { "[" "]" } (†) | ClosureTypeDec

BasicType:(†)

"byte" | "short" | "char" | "int" | "long" | "float" | "double" | "boolean"

ClosureTypeDec(*):

"Closure" ["<" Type ">"] ClosureTypeFormalParam

ClosureTypeFormalParam(*):

"(" Type { "," Type } ")"

ReferenceType:

Identifier [TypeArguments] { "." Identifier [TypeArguments] }

TypeArguments:

"<" TypeArgument { "," TypeArgument } ">"

TypeArgument(*):

Type | "?" [(extends | super) Type]

NonWildcardTypeArguments:

"<" TypeList ">"

TypeList(*):

Type { "," Type }

TypeArgumentsOrDiamond:

"<" ">" (†) | TypeArguments

NonWildcardTypeArgumentsOrDiamond:

"<" ">" | NonWildcardTypeArguments

TypeParameters:

"<" TypeParameter { "," TypeParameter } ">"

TypeParameter:

Identifier ["extends" Bound]

Bound:

ReferenceType { "&" ReferenceType }

Modifier:

Annotation | public | protected(†) | private | static | abstract | final | native(†) | synchronized(†) | transient(†) | volatile(†) | strictfp(†)

Annotations:

Annotation { Annotation }

Annotation:

"@" QualifiedIdentifier ["(" [AnnotationElement] ")"]

AnnotationElement:

ElementValuePairs | ElementValue

ElementValuePairs:

ElementValuePair { "," ElementValuePair }

ElementValuePair:

Identifier "=" ElementValue

ElementValue:

Annotation | Expression1 | ElementValueArrayInitializer

ElementValueArrayInitializer:

" [ElementValues] [","] "" "

ElementValues:

ElementValue { "," ElementValue }

ClassBody:

" { ClassBodyDeclaration } "" "

ClassBodyDeclaration(*): ";" | { Modifier } MemberDecl | static Block

MemberDecl:

MethodOrFieldDecl | "void" Identifier VoidMethodDeclaratorRest | Identifier ConstructorDeclaratorRest | GenericMethodOrConstructorDecl | ClassDeclaration | InterfaceDeclaration

MethodOrFieldDecl:

Type Identifier MethodOrFieldRest

MethodOrFieldRest:

FieldDeclaratorsRest ";" | MethodDeclaratorRest

FieldDeclaratorRest:

VariableDeclaratorRest { "," VariableDeclarator }

MethodDeclaratorRest:

FormalParameters { [] } (†) ["throws" QualifiedIdentifierList] (Block | ";")

VoidMethodDeclaratorRest:

FormalParameters ["throws" QualifiedIdentifierList] (Block | ";")

ConstructorDeclaratorRest:

FormalParameters ["throws" QualifiedIdentifierList] Block

GenericMethodOrConstructorDecl:

TypeParameters GenericMethodOrConstructorRest

GenericMethodOrConstructorRest::

(Type | “void“) Identifier MethodDeclaratorRest | Identifier ConstructorDeclaratorRest

InterfaceBody:

“ { InterfaceBodyDeclaration } ““ “

InterfaceBodyDeclaration:

“;“ | { Modifier } InterfaceMemberDecl

InterfaceMemberDecl:

InterfaceMethodOrFieldDecl | “void“ Identifier VoidInterfaceMethodDeclaratorRest | InterfaceGenericMethodDecl | ClassDeclaration(†) | InterfaceDeclaration(†)

InterfaceMethodOrFieldDecl:

Type Identifier InterfaceMethodOrFieldRest

InterfaceMethodOrFieldRest:

ConstantDeclaratorsRest(†) “;“ (†) | InterfaceMethodDeclaratorRest

ConstantDeclaratorsRest:

ConstantDeclaratorRest { “,“ ConstantDeclarator }

ConstantDeclaratorRest:

{ “[“ “]“ } = VariableInitializer

ConstantDeclarator:

Identifier ConstantDeclaratorRest

InterfaceMethodDeclaratorRest:

FormalParameters { “[“ “]“ } (†) [“throws“ QualifiedIdentifierList] “;“

VoidInterfaceMethodDeclaratorRest:

FormalParameters [“throws“ QualifiedIdentifierList] “;“

InterfaceGenericMethodDecl:

TypeParameters (Type | “void“) Identifier InterfaceMethodDeclaratorRest

FormalParameters:

“(“ [FormalParameterDecls] “)“

FormalParameterDecls:

{ VariableModifier } Type FormalParameterDeclsRest

VariableModifier(†):

“final“ | Annotation

FormalParameterDeclsRest:

VariableDeclaratorId [“,“ FormalParameterDecls] | “...“ VariableDeclaratorId

VariableDeclaratorId:

Identifier { “[“ “]“ } (†)

VariableDeclaratorId:

Identifier VariableDeclaratorRest

VariableDeclaratorRest:

{ “[“ “]“ (†) [“=“ VariableInitializer]

VariableInitializer:

ArrayInitializer | Expression

ArrayInitializer:

“ [VariableInitializer { “,“ VariableInitializer } [“]“ ,] “ ““ }

Block:

“ BlockStatements ““ “

BlockStatements:

{ BlockStatement }

BlockStatement(*):

| LocalVariableDeclarationStatement | ClassOrInterfaceDeclaration(†) | Statement

LocalVariableDeclarationStatement:

{ VariableModifier } Type VariableDeclarators “;“

Statement(*):

Block | “;“ | Identifier(†) :(†) Statement(†) | StatementExpression “;“ | IfStatement | AssertStatement “;“ | WhileStatement | DoStatement “;“ | ForStatement | BreakStatement “;“ | ContinueStatement “;“ | ReturnStatement “;“ | ThrowStatement “;“ | SynchronizedStatement | TryStatement | TypeSwitchStatement | SwitchStatement(†)

StatementExpression:

Expression

IfStatement(*):

“if“ ParExpression Block { “else if“ Block } “ else “ Block

AssertStatement:

“assert“ Expression [“:“ Expression]

WhileStatement(*):

“while“ ParExpression Block

DoStatement(*):

“do“ Block “while“ ParExpression

ForStatement(*):

“for“ “(“ ForControl “)“ Block

BreakStatement:

“break“ [Identifier] (†)

ContinueStatement:

“continue“ [Identifier] (†)

ReturnStatement:

“return“ [Expression]

ThrowStatement:

“throw“ Expression

SynchronizedStatement:

“synchronized“ ParExpression Block

TryStatement(*):

“try“ (Block | ResourceSpecification(†) Block(†)) (Catches [Finally] (†) | Finally(†))

TypeSwitchStatement(*):

“typeswitch“ “(“ Identifier “)“ “ TypeSwitchBlocks ““ “

TypeSwitchBlocks(*):

{ TypeSwitchBlock } “default“ Block

TypeSwitchBlock:

TypeSwitchLabel Block

TypeSwitchLabel:

“case“ Type

SwitchStatement:

ParExpression “ SwitchBlockStatementGroups ““ “

Catches:

CatchClause { CatchClause }

CatchClause:

catch "(" { VariableModifier } CatchType Identifier ")" Block

CatchType(*):

QualifiedIdentifier { "|" QualifiedIdentifier } (†)

Finally:

"finally" Block

ResourceSpecification:

"(" Resources [";"] ")"

Resources:

Resource { ";" Resource }

Resource:

{ VariableModifier } ReferenceType VariableDeclaratorId "=" Expression

SwitchBlockStatementGroups:

{ SwitchBlockStatementGroup }

SwitchBlockStatementGroup:

SwitchLabels BlockStatements

SwitchLabels:

SwitchLabel SwitchLabel

SwitchLabel(*):

"case" (Expression | EnumConstantName | "default") ":"

EnumConstante(*):

QuialifiedIdentifier

ForControl:

ForVarControl | ForInit(†) ";" (†) [Expression] (†) ";" (†) [ForUpdate] (†)

ForVarControl:

{ VariableModifier } Type VariableDeclaratorId ForVarControlRest

ForVarControlRest:

“:“ Expression | ForVariableDeclaratorsRest(†) “;“ (†) [Expression] (†) “;“ (†) [ForUpdate] (†)

ForVariableDeclaratorsRest:

[“=“ VariableInitializer] { “,“ VariableDeclarator }

ForInit:

StatementExpression , StatementExpression

ForUpdate:

StatementExpression { “,“ StatementExpression }

Expression:

Expression1 [AssignmentOperator Expression1]

AssignmentOperator:

“=“ | “+=“ | “-=“ | “*=“ | “/=“ | “&=“ | “|=“ | “^=“ | “%=“ (†) | “«=“ (†) | “»=“ (†) | “»>=“ (†)

Expression1:

Expression2 [Expression1Rest]

Expression1Rest†):

“?“ Expression “:“ Expression1

Expression2:

Expression3 [Expression2Rest]

Expression2Rest(*):

InfixOp Expression3 | “instanceof“ (†) Type

InfixOp:

“||“ | “&&“ | “|“ | “^“ | “&“ | “==“ | “!=“ | “<“ | “>“ | “<=“ | “>=“ | “«“ (†) | “»“ (†) | “»>“ (†) | “+“ | “-“ | “*“ | “/“ | “%“ (†)

Expression3(*):

PrefixOp Expression3 [Expression2Rest] [Expression1Rest] [AssignmentOperator Expression1] | Primary { Selector } { PostfixOp } [Expression2Rest] [Expression1Rest] [AssignmentOperator Expression1]

PrefixOp:

“++“ (†) | “–“ (†) | “!“ | “ “ (†) | “+“ | “-“

PostfixOp(†):

“++“ (†) | “–“ (†)

Primary(*):

Literal | (ParExpression | “(“ Type “)“ Expression) | “closure“ ClosureCreator | “this“ [Arguments] | “super“ SuperSuffix | “new“ Creator | “void“ [.] “class“ | Identifier { “.“ Identifier } [IdentifierSuffix] | BasicType { “[“ “]“ } “.“ “class“

:

IntegerLiteral | FloatingPointLiteral | CharacterLiteral | StringLiteral | BooleanLiteral | NullLiteral

ParExpression:

“(“ Expression “)“

ClosureCreator(*):

[ClosureEnv] [ClosureRet] FormalParameters Block

ClosureEnv(*):

“[“ ClosureEnvList “]“

ClosureEnvList(*):

{ ClosureEnvItem “;“ }

ClosureEnvItem(*):

Type Identifier “=“ Expression

ClosureRet(*):

“<“ Type “>“

Arguments:

“(“ [Expression { “,“ Expression }] “)“

SuperSuffix:

Arguments | “.“ Identifier [Arguments]

ExplicitGenericInvocation(†):

“super“ SuperSuffix | Identifier Arguments

Creator(*):

[NonWildcardTypeArguments] (†) CreatedName (ClassCreatorRest | ArrayCreatorRest(†))

CreatedName:

Identifier [TypeArgumentsOrDiamond] { “.“ Identifier [TypeArgumentsOrDiamond] }

ClassCreatorRest:

Arguments [ClassBody] (†)

ArrayCreatorRest:

"[" ["]" { "[" "]" } ArrayInitializer | Expression "]" { "[" Expression "]" } { "[" "]" }] "]"

IdentifierSuffix(*):

"[" Expression "]" { "[" Expression "]" } | Arguments "." (class | ExplicitGenericInvocation | "this" (†) | "super" (†) Arguments(†) | "new" (†) [NonWildcardTypeArguments] (†) InnerCreator(†))

InnerCreator:

Identifier [NonWildcardTypeArgumentsOrDiamond] ClassCreatorRest

Selector(*):

["." Identifier] [Arguments] | "." ExplicitGenericInvocation(†) | "." "this" (†) | "." "super" (†) SuperSuffix(†) | "." "new" [NonWildcardTypeArguments] InnerCreator(†) | "[" Expression "]" (†)

EnumBody:

"{" EnumConstants [","] [;] "}"

EnumConstants(*):

EnumConstant { "," EnumConstant }

EnumConstant:

[Annotations] Identifier [Arguments] (†) [ClassBody] (†)

EnumBodyDeclaration:

";" { ClassBodyDeclaration }

AnnotationTypeBody:

" [[ ] AnnotationTypeElementDeclarations] "" "

AnnotationTypeElementDeclarations(*):

AnnotationTypeElementDeclaration AnnotationTypeElementDeclaration

AnnotationTypeElementDeclaration:

{ Modifier } AnnotationTypeElementRest

AnnotationTypeElementRest:

Type Identifier AnnotationMethodOrConstantRest ";" | ClassDeclaration | InterfaceDeclaration | EnumDeclaration | AnnotationTypeDeclaration

AnnotationMethodOrConstantRest:

AnnotationMethodRest | ConstantDeclaratorsRest

AnnotationMethodRest:

"(" ")" | ["[" "]" | "default" ElementValue

B. TeaJays Standardbibliothek

MK

TeaJay bietet nur eine sehr minimalistische Standardbibliothek. Das ist einerseits dem noch frühen Entwicklungsstadium geschuldet, andererseits ist es auch eine Entwurfsentscheidung: Eine Lehrsprache sollte nur die nötigsten Werkzeuge mitliefern, damit Problemstellungen, die z.B. als Übungsaufgabe dienen sollen, nicht bereits gelöst, direkt vor den Augen der Lernenden, liegen. Die momentane Standardbibliothek ist aber noch nicht vollständig genug, so dass auf die direkte Benutzung von Java-APIs nicht gänzlich verzichtet werden kann. Dies sollte jedoch ein Ziel der Weiterentwicklung von TeaJay sein.

B.1. Implizit importierte Typen

Folgende Typen werden automatisch vom Compiler importiert:

java.lang :

String	Boolean	System
Object	Throwable	Exception
NullPointerException	IndexOutOfBoundsException	IllegalArgumentException
ArithmeticException	Error	IllegalStateException
RuntimeException	AssertionError	Class
Enum	NumberFormatException	OutOfMemoryError
StackOverflowError	Runtime	Thread
Runnable	CharSequence	

teajay.lang :

Collection	IO	List	Number
Sortable	TJComparable	TJException	TJIterable
TJIterator	TJObject	TJThrowable	

B.2. TeaJays Sprachbibliothek

TeaJays Sprachbibliothek enthält die notwendigsten Typen, um TeaJay-Programme schreiben zu können. Bei den meisten dieser Typen sind die Implementierungen nicht austauschbar bzw. die Implementierung ist nicht in TeaJay ausdrückbar. Hier werden nur die wichtigsten Typen kurz vorgestellt. Eine vollständigere Dokumentation der Sprachbibliothek befindet sich im Anhang.

B.2.1. List

Der Typ List ist Teil der Sprachbibliothek von TeaJay. Die Sprachbibliothek liefert eine Implementierung dieses Typs mit, jedoch ist es in TeaJay möglich eine eigene Implementierung dieses Typs zu schreiben. Der Typ List hat folgende Signatur:

```
public typedef List<T> interfaces java.io.Cloneable, Sortable<T>, teajay.lang.Collection<T> {
  List();
  List(T... values);
  List(T head, List<? extends T> tail);
  List(List<? extends T>... lists);

  void add(T value);
  void addAll(teajay.lang.TJIterable<? extends T> vals);
  T remove(Number index) throws IndexOutOfBoundsException;
  Boolean removeAll(T elem);

  T get(Number index) throws IndexOutOfBoundsException;
  void set(Number index, T value) throws IndexOutOfBoundsException;
  void insert(Number index, T value) throws IndexOutOfBoundsException;
  Number indexOf(T obj);
  Boolean contains(T elem);
  T operator_get(Number index) throws IndexOutOfBoundsException;
  void operator_set(Number index, T value) throws IndexOutOfBoundsException;

  List<T> subList(Number fromIndex) throws IndexOutOfBoundsException;
  List<T> subList(Number fromIndex, Number endIndex) throws IndexOutOfBoundsException;

  void clear();
  Number size();

  Boolean isEmpty();

  List<T> clone();
  TJIterator<T> iterator();
}
```

B.2.2. Number

Der Typ Number dient in TeaJay der Darstellung rationaler Zahlen. Er hat folgende Signatur:

```
public final typedef Number interfaces java.lang.Comparable<Number>, java.io.Serializable, ↩
    ↪TJComparable<Number> {
  Number();
  Number(Number n);

  Number add(Number a);
  Number mult(Number a);
  Number sub(Number a);
  Number div(Number a) throws ArithmeticException;
  Number mod(Number n);
  Number pow(Number exponent);

  static Number operator_sub(Number a);
  static Number operator_sub(Number a, Number b);
  static Number operator_add(Number a, Number b);
  static Number operator_mult(Number a, Number b);
  static Number operator_div(Number a, Number b) throws ArithmeticException;

  static Boolean operator_less(Number a, Number b);
  static Boolean operator_leq(Number a, Number b);
  static Boolean operator_greater(Number a, Number b);
  static Boolean operator_geq(Number a, Number b);

  static Number parseNumber(String str) throws NumberFormatException;
  static Number parseInteger(String str) throws NumberFormatException;
  static Number parseInteger(String str, Number radix) throws NumberFormatException;
  static Number parseTJNumber(String number) throws NumberFormatException;

  Number and(Number other);
  Number or(Number other);
  Number xor(Number other);
  Number shiftLeft(Number bitCount);
  Number shiftRight(Number bitCount);

  Number abs();
  Number not();
  Number floor();
  Number ceil();
  Number toInteger();
  Number negate();
  Number invert() throws ArithmeticException;

  Number numerator();
```

```
43    Number denominator();
44    Number signum();
45
46    Boolean isInteger();
47    Boolean isNegative();
48    Boolean divides(Number a) throws ArithmeticException;
49    Boolean operator_eq(Object obj);
50
51    Number round();
52    Number round(Number precision) throws ArithmeticException;
53    Number round(Number precision, java.math.RoundingMode mode) throws ArithmeticException;
54
55    String toDecimalString();
56    String toDecimalString(Number precision);
57    String toDecimalString(Number precision, java.math.RoundingMode rmode);
58    String toEngineeringString();
59    String toEngineeringString(Number precison);
60    String toEngineeringString(Number precison, java.math.RoundingMode rmode);
61    String toFractionString();
62  }
```

Wie man erkennen kann, bietet Number auch Bitweise-Operationen. Diese operieren nur auf ganzen Zahlen und werfen eine Ausnahme, wenn die fragliche Zahl keine ganze Zahl ist. Diese Operationen nehmen an, dass die Zahl im Zweierkomplement dargestellt wird.

B.2.3. String

Der TeaJay-Typ String wird direkt auf die Java-Klasse java.lang.String abgebildet. Dadurch implementiert String zwar Comparable<String> aber nicht TJComparable<String>. Auch verliert java.lang.String einige Methoden, die in TeaJay nicht ohne die Java-Kompatibilitätsschicht aufgerufen werden können[1], während andere auf äquivalente TeaJay-Methoden umgeleitet werden. Zum Beispiel wird char charAt(int index) auf String charAt(Number index) umgeleitet.

```
1   public final typedef String extends java.lang.Object interfaces java.io.Serializable, java.lang↩
        ↪.Comparable<String>, CharSequence {
2     String(String str);
3
4       String charAt(Number index);
5
6       String concat(String str);
7       static operator_add(String a, Object b);
8       static operator_add(Object a, String b);
9
10      Boolean contains(CharSequence s);
11      Boolean contentEquals(StringBuffer sb);
12      Boolean contentEquals(CharSequence cs);
13      Boolean endsWith(String suffix);
14      Boolean equalsIgnoreCase(String anotherString);
15
16      Number indexOf(String str);
17      Number indexOf(String str, Number fromIndex);
18
19      String intern();
20
21      Boolean isEmpty();
22
23      Number lastIndexOf(String str);
24      Number lastIndexOf(String str, Number fromIndex);
25      Number length();
26
27      Boolean matches(String regex);
28      String replace(CharSequence target, CharSequence replacement);
29      String replaceAll(String regex, String replacement);
30      String replaceFirst(String regex, String replacement);
31
32      List<String> split(String regex, Number limit);
33      List<String> split(String regex);
34
35      Boolean startsWith(String prefix, Number toffset);
36      Boolean startsWith(String prefix);
37      String substring(Number beginIndex);
38      String substring(Number beginIndex, Number endIndex);
39
40      String toLowerCase(java.util.Locale locale);
```

[1] Diese Methoden können immer noch mithilfe der Java-Kompatibilitätsschicht aufgerufen werden, sollen jedoch nicht Teil des TeaJay-Typs String sein.

```
41      String toLowerCase();
42      String toUpperCase(java.util.Locale locale);
43      String toUpperCase();
44      String trim();
45
46      static Boolean operator_less(String a, String b);
47      static Boolean operator_leq(String a, String b);
48      static Boolean operator_greater(String a, String b);
49      static Boolean operator_geq(String a, String b);
50      Boolean operator_eq(Object o);
51
52      public List<String> toCharacterList();
53      public TJIterator<String> iterator();
```

B.2.4. CharSequence

Das interface CharSequence hat aus Sicht von TeaJay folgende Signatur:

```
1   public interface CharSequence {
2     Number length();
3     String charAt(Number index);
4     CharSequence subSequence(Number start, Number end);
5   }
```

TeaJay-Typen können dieses interface jedoch nicht erfüllen, da die tatsächlichen Methodensignaturen Java-Primitive enthalten und somit nicht von TeaJay-Code dargestellt werden können.

B.2.5. Boolean

Der TeaJay-Typ Boolean wird, ähnlich wie String, auf die Java-Klasse java.lang.Boolean abgebildet:

```
1   public final typedef Boolean extends java.lang.Object interfaces java.io.Serializable, java.↩
        ↪lang.Comparable<Boolean> {
2     Boolean(String);
3     static Boolean parseBoolean(String s);
4     static Boolean valueOf(String s);
5     Boolean operator_eq(Object o);
6     static Boolean getBoolean(String propertyName);
7   }
```

B.2.6. TJObject

TJObject ist Supertyp aller in TeaJay definierten Typen. Dies gilt jedoch nicht für einige eingebaute Typen wie z.B. String oder Boolean. TJObject ist daher eher ein Implementierungsdetail und stellt die überschreibbare Methode tjHashCode() bereit. Man sollte daher nie direkt TJObject verwenden und auch nicht tjHashCode() direkt aufrufen, sondern TJObjects.hashCode(Object) (weist auch null einen Hashwert zu) oder hashCode() verwenden.

```
1   public typedef TJObject extends java.lang.Object {
2     TJObject();
3
4     Number tjHashCode();
5
6     Number hashCode(); //nicht überschreibbar, existiert nur virtuell
7
8     Boolean operator_eq(Object o);
9
10    String toString();
11  }
```

B.2.7. System

System erlaubt rudimentäre Kommunikation mit der JVM und der Prozessumgebung. System wird auf java.lang.System abgebildet und, sofern notwendig, werden die Methodenaufrufe, wie im Fall von String, umgeleitet:

```
1  public typedef System extends java.lang.Object {
2    static void setIn(java.io.InputStream in);
3    static void setOut(java.io.PrintStream out);
4    static void setErr(java.io.PrintStream out);
5
6    static java.io.Console console();
7
8    static Number currentTimeMillis();
9    static Number nanoTime();
10
11   static void arraycopy(JavaArray<?> src, Number srcPos, JavaArray<?> dest, Number destPos, ↩
       ↪Number len);
12
13   static Number identityHashCode(Object x);
14
15   static Properties getProperties();
16   static void setProperties(Properties props);
17   static String lineSeparator();
18   static String getProperty(String key);
19   static String getProperty(String key, String def);
20   static String setProperty(String key, String value);
21   static String clearProperty(String key);
22   static String getenv(String name);
23   static java.util.Map<String,String> getenv();
24   static void exit(Number status)
25 }
```

B.2.8. TJThrowable und TJException

Da es in TeaJay nicht möglich ist, direkt von java.lang.Throwable oder java.lang.-Exception zu erben, gibt es in TeaJay TJThrowable und TJException. Sie ermöglichen es in TeaJay eigene Ausnahmetypen zu definieren.

TJThrowable :

```
1  public typedef TJThrowable extends java.lang.Throwable {
2    TJThrowable();
3    TJThrowable(String message);
4    TJThrowable(String message, Throwable cause);
5    TJThrowable(Throwable cause);
6
7    Number tjHashCode();
8      Boolean operator_eq(Object obj);
9  }
```

TJException :

```
1  public typedef TJException extends java.lang.Exception {
2    TJException();
3    TJException(String message);
4    TJException(String message, Throwable cause);
5    TJException(Throwable cause);
6
7    Number tjHashCode();
8      Boolean operator_eq(Object obj);
9  }
```

B.2.9. RandomAccess

Das interface RandomAccess<T> definiert Operationen für wahlfreien Zugriff über Indizes. Es hat folgende Signatur:

```
1 public interface RandomAccess<T> {
2     T get(Number index);
3     void set(Number index, T val);
4     Number size();
5 }
```

B.2.10. TJComparable

Das interface TJComparable<T> lehnt sich an das aus Java bekannte interface java.lang.Comparable<T> an und bietet dieselbe Funktionalität. Mit TJComparable ist es möglich Vergleichbarkeit in einer generischen Typsignatur zu fordern: <T **extends** TJComparable<? **super** T>>. Dieses interface ist notwendig, da man keine generische Signatur angeben kann, die fordert, dass ein Typ die Operatoren > und < überschreibt.

```
1 public interface TJComparable<T> {
2     Number tjCompareTo(T other);
3 }
```

B.2.11. TJIterable

TJIterable zeigt an, dass ein Typ iterierbar ist. Ein Typ der TJIterable erfüllt kann in der for-Schleife verwendet werden.

```
1 public interface TJIterable<T> {
2     TJIterator<T> iterator();
3 }
```

B.2.12. TJIterator

TJIterator definiert eine Schnittstelle, welche es ermöglicht über einen beliebigen Typ T zu iterieren.

```
1 public interface TJIterator<T> {
2     Boolean hasNext();
3     T next() throws IllegalStateException;
4     void remove();
5 }
```

B.2.13. Collection

Collection definiert Operationen für den Umgang mit Ansammlungen von Objekten.

```
1 public interface Collection<T> extends TJIterable<T> {
2     Boolean contains(T elem);
3     Boolean removeAll(T elem);
4     void add(T elem);
5     Number size();
6     void clear();
7 }
```

B.2.14. IO

Der Typ IO erlaubt unter anderem die Aus- und Eingabe von Zeichenketten auf der Systemkonsole. Er bietet auch TeaJay-kompatible Methoden, um vorzeichenlose Bytes von einem java.io.InputStream zu lesen bzw. in einen java.io.OutputStream zu schreiben.

```
1  public final typedef IO {
2      static void println(Object obj);
3      static void printlnErr(Object obj);
4      static void print(Object obj);
5      static void printErr(Object obj);
6
7      static String readLine();
8
9      static void writeBytes(java.io.OutputStream out, List<Number> bytes) throws IOException;
10     static void writeBytes(java.io.OutputStream out, List<Number> bytes, Number offset, Number ↩
           ↪len) throws IOException;
11     static void writeBytes(java.io.OutputStream out, Number... bytes) throws IOException;
12
13     static Number readBytes(java.io.InputStream in, List<Number> out, Number len) throws ↩
           ↪IOException;
14     static Number readBytes(java.io.InputStream in, List<Number> out, Number offset, Number len↩
           ↪) throws IOException;
15 }
```

B.3. teajay.util

Im Paket teajay.util finden sich verschiedene Werkzeuge. Hier wird nur eine Übersicht der Typen in diesem Paket gegeben. Eine Dokumentation findet sich im Anhang.

Math stellt die wichtigsten numerischen Funktionen bereit.

Characters bietet Methoden um TeaJay-Buchstaben zu Zeichenketten umzuwandeln.

ArrayUtils bietet Methoden um Java-Arrays in List und umgekehrt umzuwandeln.

RandomGenerator bietet Funktionalität um Zufallszahlen generieren zu können.

Real dient der schnellen numerischen Berechnung und stellt eine IEEE754 Fließkommazahl mit doppelter Genauigkeit dar. (siehe auch double in Java)

Sorter ermöglicht das Sortieren von Typen, welche RandomAccess erfüllen.

Strings ermöglicht das Umwandeln von Listen mit Bytes in Zeichenketten.

TJObjects bietet Werkzeuge für den Umgang mit Referenzen.

B.4. java.io.File

Die Java-Klasse java.io.File wird auf ähnliche Art und Weise wie java.lang.String zu einem TeaJay-Typ gemacht. Damit besitzt TeaJay einen Typ zur Darstellung von Dateien. Momentan fehlt TeaJay aber eine Bibliothek für die Ein- und Ausgabe auf Dateien. Ein Entwickler muss sich daher bei der Java-API bedienen oder eine in TeaJay geschriebene Bibliothek entwickeln bzw. nutzen. In Kapitel 4 wird eine solche API exemplarisch entwickelt.

C. Datenträger

MB
MK

Anhang C befindet sich auf dem beiliegenden Datenträger und enthält folgendes:

- den Quelltext aller Projekte
- TeaJay-Beispielprogramme
- Scripte zum Installieren, Ausführen sowie Kompilieren von TeaJay
- javadoc-Dokumentationen
- einige Benutzte Quellen (als PDF)

Quellcodeverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Literaturverzeichnis

[ASL+08] Aho, Alfred V, Ravi Sethi, Monica S Lam et al.: *Compiler: Prinzipien, Techniken und Werkzeuge*. Pearson Deutschland GmbH, 2008.

[BH98] Bauer, Bernhard und Riita Höllerer: *Übersetzung Objektorientierter Programmiersprachen: Konzepte, Abstrakte Maschinen und Praktikum'Java-Compiler'*. Springer DE, 1998.

[BSW10] Bethlen, Gregor, Peter H Schmitt und Benjamin Weiss: *Funktionsabschlüsse in Dynamischer Logik*. 2010.

[BW09] Biancuzzi, Frederico und Shane Warden: *Visionäre der Programmierung - Die Sprachen und ihre Schöpfer*. O'REILLY, 2009.

[Bö95] Böhme, P.: *Strukturierte Programmierung*. `http://www2.informatik.uni-halle.de/lehre/prog/strukt_prog.html`, 1995. Abgerufen: 18. November 2013.

[Cat13] Catambay, Bill: *The Pascal Programming Language*. `http://www.pascal-central.com/ppl/chapter3.html#Unextended`, November 2013.

[cey14] *Ceylon: Welcome to Ceylon*. `http://www.ceylon-lang.org/`, Januar 2014.

[Cla11] Clausing, Achim: *Programmiersprachen - Konzepte, Strukturen und Implementierung in Java*. Spektrum, 2011.

[Coo] Cook, William R.: *Object-Oriented Programming Versus Abstract Data Types*. In: *Proc. of the REX Workshop/School on the Foundations of Object-Oriented Languages, LNCS 173*, Seiten 151–178. Springer Verlag.

[Cur99] *Richtlinien und Lehrpläne für die Sekundarstufe II – Gymnasium/Gesamtschule in Nordrhein-Westfalen - Informatik*. Ministerium für Schule und Weiterbildung, Wissenschaft und Forschung des Landes Nordrhein-Westfalen, 1999.

[Dij63] Dijkstra, EW: *Making a Translator for ALGOL 60*. Annual review in automatic programming, 3:347–356, 1963.

[EG96] E. Gamma, R. Helm, R. Johnson J. Vlissides: *Entwurfsmuster - Elemente wiederverwendbarer objektorientierter Software*. Addison-Wesley, 1996.

[Eif13] *Eiffel (Programmiersprache).* `http://de.wikipedia.org/wiki/Eiffel_(Programmiersprache)`, Dezember 2013. Abgerufen: 17.01.14.

[for] *The For-Each Loop.* `http://docs.oracle.com/javase/1.5.0/docs/guide/language/foreach.html`. Abgerufen: 15. November 2013.

[GJS+12] Gosling, James, Bill Joy, Guy Steele, Gilad Bracha und Alex Buckley: *The Java™ Language Specification Java SE 7 Edition.* July 2012.

[Goe97] Goedicke, Michael: *Java in der Programmierausbildung: Konzept und erste Erfahrungen.* Technischer Bericht, Unviersität Duisburg-Essen, 1997.

[Gut77] Guttag, John: *Abstract data types and the development of data structures.* Communications of the ACM, 20(6):396–404, 1977.

[H.K98] H.Ketz., K.Hug: *Informatik-Grundausbildung für Ingenieure- Hochschuldidaktische Betrachtung und Erfahrungsbericht.* In: *Proc. GI-Fachtagung 98.* Springer Verlag, 1998.

[jav13] *Java Platform SE 7.* `http://docs.oracle.com/javase/7/docs/api/`, 2013. Abgerufen 30. Dezember 2013.

[Jia] Jiang, Xiaoyi: *Informatik II – Datenstrukturen und Algorithmen Kapitel 11.*

[KRD13] Klein, Gerwin, Steve Rowe und Régis Décamps: *JFlex - The Fast Scanner Generator for Java.* `http://www.jflex.de`, Januar 2013. Abgerufen: 5. Januar 2014.

[Leh10] *Lehrplan Informatik - Gymnasialer Bildungsgang - Gymnasiale Oberstufe.* Hessisches Kultusministerium, 2010.

[LN] Loser, Roland und O. Nierstrasz: *Objekt-orientierter Compilerentwurf.*

[LYBB12] Lindholm, Tim, Frank Yellin, Gilad Bracha und Alex Buckley: *The Java™ Virtual Machine Specification, Java SE 7 Edition*, July 2012.

[MC13] Mike Cowlishaw, Joseph D. Darcy: *MathContext.java.* `http://docs.oracle.com/javase/7/docs/api/java/math/MathContext.html`, November 2013.

[Mey92] Meyer, Bertrand: *Applying "design by contract".* Computer, 25(10):40–51, 1992.

[Mey96] Meyer, Bertrand: *The many faces of inheritance: A taxonomy of taxonomy.* Computer, 29(5):105–108, 1996.

[MK06] Maier-Komor, Thomas: *Methoden der Metaprogrammierung zur Rekonfiguration von Software eingebetteter Systeme*. Technical University Munich, 2006.

[Moo00] Moore, Robert C: *Removing left recursion from context-free grammars*. In: *Proceedings of the 1st North American chapter of the Association for Computational Linguistics conference*, Seiten 249–255. Association for Computational Linguistics, 2000.

[Mor] Morrison, J.Paul: *Flow-Based Programming*. `http://jpaulmorrison.com/fbp/`.

[Off13] *Off-by-one-Error*. `http://de.wikipedia.org/wiki/Off-by-one-Error`, November 2013. Stand: 15. November 2013, 01:25 Uhr.

[Ola13] Olaf: *Unit/de*. `http://wiki.freepascal.org/Unit/de`, November 2013. Abgerufen: 15.01.2014.

[ora13] *Lesson: Generics*. `http://docs.oracle.com/javase/tutorial/java/generics/`, Dezember 2013.

[Pin] Pingel, Steffen: *Abstrakte syntaxbäume*.

[PNW14] Paulson, Larry, Tobias Nipkow und Makarius Wenzel: *Isabelle*. `http://isabelle.in.tum.de/`, Januar 2014.

[Pun07] Puntigam, Franz: *Objektorientierte Programmierung*, 2007.

[Sch95] Schwill, Andreas: *Programmierstile im Anfangsunterricht*. In: *Innovative Konzepte für die Ausbildung*, Seiten 178–187. Springer, 1995.

[SDRS12] Solodkyy, Yuriy, Gabriel Dos Reis und Bjarne Stroustrup: *Open and efficient type switch for C++*. In: *Proceedings of the ACM international conference on Object oriented programming systems languages and applications*, Seiten 963–982. ACM, 2012.

[Sof] *SoftReference*. `http://docs.oracle.com/javase/7/docs/api/java/lang/ref/SoftReference.html`. Abgerufen: 02.02.14.

[SR03] Scheffer, Arne und Ingo Rau: *Implementierung einer Pattern-Matching-basierten Programmiersprache*. Diplomarbeit, Westfälische Wilhelms-Universität Münster, September 2003.

[Str96] Strom, R.: *Do Programmers Need Seat Belts?* In: *SIGPLAN Notices 31*, Seiten 6–7, 1996.

[TIO14] *TIOBE-Index*. `http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html`, Januar 2014. Abgerufen: 06.01.2014.

[TJJV04] Truyen, Eddy, Wouter Joosen, Bo Nørregaard Jørgensen und Pierre Verbaeten: *A generalization and solution to the common ancestor dilemma problem in delegation-based object systems*. In: *Dynamic Aspects Workshop (DAW04)*, Band 6, 2004.

[uni14] *Unicode Consortium*. `http://www.unicode.org/`, 2014.

[www13] *Free Pascal*. `http://www.freepascal.org`, November 2013.

[Zen14] *Vorgaben zu den unterrichtlichen Voraussetzungen für die schriftlichen Prüfungen im Abitur in der gymnasialen Oberstufe im Jahr 2014*. `http://www.standardsicherung.schulministerium.nrw.de/abitur-gost/fach.php?fach=15`, Januar 2014. Abgerufen: 08.01.2014.