JOHANNES KEPLER
UNIVERSITÄT LINZ

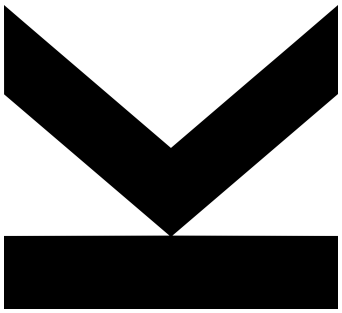
Eingereicht von
Felix Schmid
11806713

Angefertigt am
Institut für Systemsoftware

Beurteiler
Prof. Dr. Dr. h.c.
Hanspeter Mössenböck

August 2023

Erweiterung von MicroJava um Objektorientierung und Ausnahmebehandlung



Bachelorarbeit

zur Erlangung des akademischen Grades

Bachelor of Science

im Bachelorstudium

Informatik

JOHANNES KEPLER
UNIVERSITÄT LINZ
Altenbergerstraße 69
4040 Linz, Österreich
www.jku.at
DVR 0093696

Kurzfassung

Die kleine, von Java inspirierte Sprache 'MicroJava' wird hauptsächlich für die Lehrveranstaltung Compilerbau an der JKU verwendet. Derzeit bietet diese Sprache jedoch nur einen Bruchteil der Funktionen von Java, da nur wenige grundlegende Datentypen und Ausdrücke zur Verfügung stehen. Dabei fehlen vor allem objektorientierte Konzepte und Möglichkeiten der Ausnahmebehandlung, welche beide wichtige Merkmale von Java darstellen. In dieser Arbeit wird MicroJava um vereinfachte Ausführungen dieser Funktionen ergänzt. Konkret werden Vererbung, Objektmethoden mit dynamischer Bindung, Typumwandlungen, Typtests und das Werfen und Fangen von Ausnahmen hinzugefügt.

Um diese Ziele zu erreichen, wird die Arbeit Schritt für Schritt alle notwendigen Änderungen durchbesprechen. Dabei muss die Syntax der Sprache und der Objektdatei erweitert werden und es müssen einige Modifizierungen sowohl im Compiler als auch in der virtuellen Maschine von MicroJava (μJVM) erfolgen. Zuerst muss die Symbolliste des Compilers erweitert werden, um neue Konzepte wie Vererbung zu unterstützen. Des Weiteren muss der Parser für die neue Syntax und die neuen semantischen Kontextbedingungen angepasst werden. Außerdem gibt es nun neue Bytecodes, welche vom Codegenerator erzeugt werden, welche zum Beispiel Typtests und das Werfen von Ausnahmen ermöglichen. Diese neuen Bytecodes müssen folglich auch in der μJVM unterstützt und implementiert werden.

Zum Schluss wird sich diese Arbeit auch mit dem Testen dieser neuen Funktionen beschäftigen, damit eine hohe Zuversicht in deren korrekter Implementierung erreicht werden kann. Damit dies gelingt werden sowohl manuell als auch automatisch erstellte Testfälle benutzt, um systematisch die korrekte Verhaltensweise in vielen verschiedenen Situationen zu gewährleisten.

Abstract

The small, Java inspired language called 'MicroJava' is primarily used to teach compiler construction in the corresponding JKU lecture. However, it currently only offers very limited functionality compared to Java, as only the most basic data types and statements are available. Most important, it is missing object-orientation and exception handling concepts at the moment, which are key features of Java. In this thesis, a simplified version of those features will be added to MicroJava. More specifically, inheritance, object-methods with dynamic binding, type casts, type tests and throwing and catching exceptions will be added.

To achieve this goal, this thesis will walk through all the necessary changes in each part: the language and object file syntax, the MicroJava Compiler and the MicroJava Virtual Machine (μJVM). First, the compiler requires an expanded symbol table to account for new concepts such as inheritance. Furthermore, the parser has to deal with the new syntax and semantic conditions. There are also new bytecodes being generated by the compiler, which enable type tests and throwing exceptions for example. These new bytecodes must then also be implemented and handled in the μJVM.

Finally, this thesis will also deal with testing every new feature, so that a high level of confidence in the correctness of the implementation can be achieved. To do so, both manual and automatically generated test cases will be used to systematically confirm the correct behaviour in many different situations.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

1 Einleitung

Die Sprache MicroJava ist eine einfache, an Java angelehnte Programmiersprache, die am Institut für Systemsoftware an der JKU zum Unterrichten von Compilerbau eingesetzt wird. Dabei implementieren alle Studierende im Laufe der Vorlesung und der dazugehörigen Übung ihren eigenen Compiler für MicroJava, wobei die grobe Struktur des Parsers, Scanners, Codegenerators und weiterer Hilfsklassen bereits vorgegeben ist. Konkret handelt es sich um einen Einpass-Compiler, welcher die Quelldatei einliest und zugleich das Zielprogramm erzeugt, welches im Falle von MicroJava eine Objektdatei mit wiederum an Java angelehnten Bytecode ist. Ausgeführt werden diese Objektdateien durch die virtuelle Maschine von MicroJava (μJVM), die bereits vorgegeben ist. Der MicroJava Compiler wird dabei selbst mit Java implementiert, da dies die im Informatikstudium an der JKU am häufigsten eingesetzte Sprache ist. Die Musterlösung des MicroJava Compilers dient dabei gemeinsam mit dem Stoff der Vorlesung als Ausgangspunkt für diese Arbeit.

Im Zuge eines Ausblicks auf weiterführende Themen am Ende der Compilerbau Vorlesung werden die Implementierungen von fortgeschrittenen Funktionen für MicroJava, wie überladene Methoden oder getrennte Übersetzung von mehreren Quelldateien, skizziert. Zwei dieser dort angesprochenen Funktionen, Objektorientierung und Ausnahmebehandlung, werden nun konkret im Zuge dieser Arbeit implementiert. Dafür sind weitreichende Änderungen in allen Teilen des Compilers und auch der μJVM notwendig, welche hier Schritt für Schritt beschrieben werden. Die Resultate dieser konkreten Implementierung dienen schlussendlich auch der Ausbesserung und Erweiterung der in den Vorlesungsfolien skizzierten Vorschläge.

2 Überblick über MicroJava

Bevor die Änderungen und Erweiterungen des Compilers beschrieben werden, soll in diesem Kapitel noch ein kurzer Überblick über den derzeitigen Stand der Sprache MicroJava gegeben werden. Wie bereits angesprochen, ist MicroJava eine recht einfache Sprache, welche sich an Java orientiert. Trotz der Vereinfachung sind die wichtigsten Konzepte von Programmiersprachen vorhanden.

Als Datentypen stehen die Basistypen `int` und `char`, Klassen und eindimensionale Arrays zur Verfügung. Arrays können dabei auch von Klassen erstellt werden und in Klassen können Felder von allen besprochenen Datentypen angelegt werden, jedoch sind dort noch keine Methodendefinitionen erlaubt. Auch Vererbung von Klassen wird noch nicht unterstützt. Am Beginn des Programms können neben Klassen auch noch statische Variablen und `int`- oder `char`-Konstanten definiert werden. Danach folgen Methodendefinitionen, welche beliebige Parameter und lokale Variablen besitzen können. Für Rückgabewerte dürfen hingegen nur die Basistypen `int` und `char` verwendet werden. Als Kontrollstrukturen existieren `if`- und `while`-Ausdrücke. Weitere Statements für die Interaktion mit der Befehlszeile sind `read` und `print`. Außerdem gibt es drei vordefinierte Methoden (`ord`, `chr` & `len`), welche zum Umwandeln der Basistypen bzw. zum Ermitteln von Arraylängen genutzt werden können.

Kompilationsziel der Sprache ist eine Objektdatei, welche neben einigen Metadaten vor allem den generierten Bytecode enthält. Die Art und Funktionsweise der Instruktionen sind dabei eine vereinfachte Versionen des Java Bytecodes. Beide Ausführungen basieren dabei auf dem Prinzip einer Stackmaschine, es existieren im Sinne der Sprache also keine Prozessor-Register. Stattdessen können etwaige Werte auf den sogenannten Expressionstack abgelegt und wieder von oben abgenommen werden. Dieses Prinzip soll mit dem Beispiel in Abb. 2.1 verdeutlicht werden, die Instruktion `add` nimmt z.B. als Operanden die zwei obersten Werte des Expressionstacks und gibt das Ergebnis wieder dahin zurück.

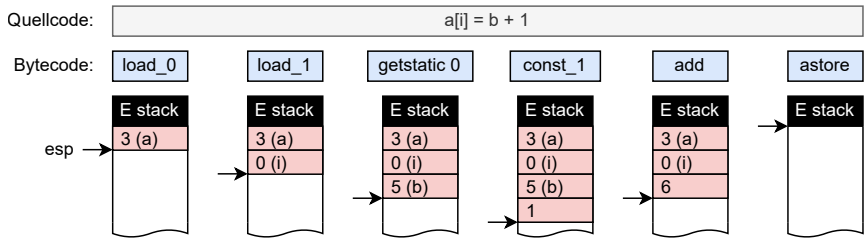


Abbildung 2.1: Veränderung des Expressionstacks für ein Codebeispiel.

Ausgeführt werden diese Instruktionen in der bereits erwähnten μJVM. Diese virtuelle Maschine ist wie auch der Compiler in Java geschrieben und ebenfalls sehr einfach gehalten. So gibt es zum Beispiel keinen Garbage-Collector, was bedeutet, dass alle erzeugten Objekte für die gesamte Dauer des Programms bestehen bleiben. Auch die Bytecode Instruktionen werden ohne weitere Optimierungen - wie etwa der Just-in-time-Kompilierung in Java - eingelesen, interpretiert und ausgeführt. Der dafür zuständige Interpreter macht dadurch den Großteil der μJVM aus. Deswegen, und auch wegen des leichter zu lesenden Namens, wird im Rest dieser Arbeit daher immer vom 'Interpreter' die Rede sein.

Dieser Interpreter hat zusätzlich zu dem bereits angesprochenen **Expressionstack** auch noch vier weitere wichtige Speicherbereiche. Der **Codebereich** enthält dabei den gesamten eingelesenen Bytecode des Programms und wird mithilfe des Programmzeigers verarbeitet. Der **Datenbereich** ist für die Speicherung aller statischen Variablen zuständig, seine Größe ist fix durch die Metadaten der Objektdatei festgelegt. Im **Heap** werden nacheinander alle erstellten Objekte gespeichert. Als letzter Speicherbereich existiert noch der **Methodenstack**, wo die lokalen Variablen, Parameter und Rückkehradressen von Methoden gespeichert werden. Des Weiteren wird bei Methodenaufrufen ein dynamischer Link auf den vorherigen Methodenframe abgespeichert, wodurch man mit zusätzlicher Hilfe von Frame- und Stackpointer den Methodenstack korrekt auf- und abbauen kann.

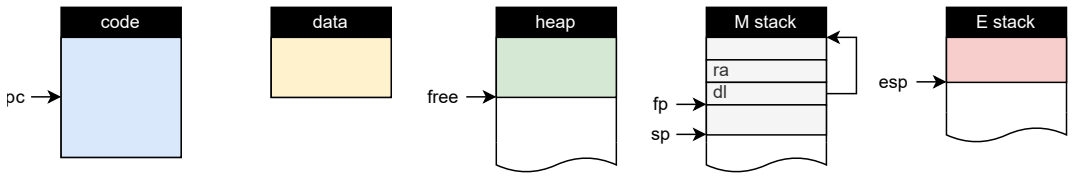


Abbildung 2.2: Speicherlayout im Interpreter von MicroJava.

Ein detaillierter Überblick über die Grammatik und alle Bytecode-Instruktionen von MicroJava sind auf der Institutswebsite in Appendix A & B des Handouts zu finden.[1]

3 Objektorientierung

Die Klassen von MicroJava sind derzeit rein zusammengesetzte Datentypen, die keine objektorientierte Merkmale wie Vererbung oder Methoden erlauben – ähnlich wie ein `struct` in C. Da Objektorientierung aber ein wichtiges Konzept in Java darstellt, soll nun auch MicroJava um dieses erweitert werden. Konkret wird nun erlaubt, Objektmethoden in Klassen zu definieren, welche direkt auf die Felder der Klasse Zugriff haben. Zusätzlich können Klassen nun von anderen Klassen erben, wobei Unterklassen alle Felder und Methoden der Oberklassen erhalten. Methoden können dabei auch überschrieben werden, wodurch es zur Laufzeit zu virtuellen Methodenaufrufen kommt, bei denen die Methode nicht anhand des statischen, sondern des dynamischen Typs des Objekts entschieden wird. Des Weiteren wird noch die Möglichkeit hinzugefügt, den Typ von Objekten wie in Java mit dem Schlüsselwort `instanceof` zu überprüfen und eine Typumwandlung (type cast) durchzuführen.

3.1 Syntax und Kontextbedingungen

Um diese neuen Funktionen der Objektorientierung zu erlauben, muss zunächst die Grammatik und damit die Syntax der Sprache angepasst werden. Es existieren nun zwei neue Schlüsselwörter, die im Scanner und als Tokentyp definiert werden müssen: `extends` und `instanceof`. Zusätzlich wird der Scanner um eine Möglichkeit erweitert, ein Token vorauszuschauen, ohne dies zu verbrauchen. Diese Funktion war im vorgegebenen Scanner noch nicht vorhanden, da dieser aber sehr ähnlich zu dem Scanner des Compilergenerators Coco/R aufgebaut wurde, konnte die Implementierung für `peek()` von dort übernommen werden.[2] Diese peek-Funktion des Scanners wird später im Parser benötigt, da die neue Grammatik von MicroJava nicht mehr LL(1) konform ist.

Um Vererbungen und Objektmethoden syntaktisch zu erlauben, muss die Produktion *ClassDecl* in der Grammatik erweitert werden. Klassen können nun optional von einer anderen Klasse erben und es sind neben Variablendeklarationen auch Methodendeklarationen erlaubt. Durch die Änderungen ergeben sich 3 neue Kontextbedingungen. Der blaue Text stellt jeweils die Änderungen gegenüber der ursprünglichen Produktion dar:

```
ClassDecl = "class" ident ["extends" ident] "{" {VarDecl | MethodDecl} "}".
```

Kontextbedingungen:

- Der optionale *ident* muss ein Typ sein und eine Klasse bezeichnen.
- Es dürfen keine zyklischen Abhängigkeiten in der Vererbung entstehen.
- Überschriebene Methoden müssen den gleichen Rückgabewert und die gleichen Parameter besitzen.

Durch diese geänderte Produktion ist die Grammatik nicht mehr LL(1) konform, da sowohl *VarDecl* als auch *MethodDecl* mit den selben Tokentypen beginnen, wenn die Methode keinen Rückgabewert besitzt, wie man anhand der Grammatik erkennen kann:

```
VarDecl = Type ident {"," ident} ";".
MethodDecl = (Type | "void") ident "(" ...
```

Um dieses Problem zu lösen, benutzt der Parser die Methode `peek()` des Scanners um die Symbole *Type* und *ident* zu überspringen. Anhand des nächsten Symbols wird dann entschieden, welche der zwei Produktionen betreten wird. Handelt es sich um das Symbol "(", so wird die Produktion *MethodDecl* verwendet, ansonsten geht der Parser zu *VarDecl*.

Als nächstes kann die Produktion *CondFact* nun zusätzlich zu Vergleichsoperationen einen Typtest mit folgenden Kontextbedingungen darstellen:

```
CondFact = Expr0 (Relop Expr1 | "instanceof" Type).
```

Kontextbedingungen:

- Der Typ von $Expr_0$ muss im Fall von *instanceof* eine Klasse sein.
- *Type* muss eine Klasse bezeichnen.
- Eine der zwei Klassen muss ein Untertyp oder gleich der anderen sein.

Zuletzt muss noch eine Alternative der Produktion *Factor* angepasst werden, damit eine Typumwandlung möglich ist - es ergeben sich dabei die gleichen Kontextbedingungen wie vorher. Die anderen Alternativen von *Factor* werden nicht verändert und wurden hier deshalb ausgelassen:

`Factor = ... | "(" (Expr`$_0$ `")" | Type ")" Expr`$_1$`).`

Kontextbedingungen:

- Der Typ von $Expr_1$ muss eine Klasse sein.
- *Type* muss eine Klasse bezeichnen.
- Eine der zwei Klassen muss ein Untertyp oder gleich der anderen sein.

Auch hier entsteht wieder ein LL(1) Konflikt, da sowohl $Expr_0$ als auch *Type* mit dem Symbol *ident* beginnen können. Hier kann der Konflikt durch semantische Information aufgelöst werden. Handelt es sich bei *ident* um einen Typ, so wird die Cast-Alternative verwendet, ansonsten geht der Parser von einem geklammerten Ausdruck aus. Bei einem Cast wird auch der eigentlich überflüssige Upcast zugelassen, bei dem ein Typ auf einen Obertyp umgewandelt wird, da dies in Java ebenfalls erlaubt ist.[3]

Zusätzlich muss noch eine bestehende Kontextbedingungen leicht angepasst werden:

`Designator = ident`$_0$ `{"." ident`$_1$ `| "[" Expr "]"}.`

Geänderte Kontextbedingung:

- $ident_1$ muss ein Feld oder eine Methode von $ident_0$ sein.

Aufgrund der neuen oder geänderten Bytecodes, die in späteren Kapiteln näher erläutert werden und teilweise Parameter mit limitierter Größe von 8 oder 16 Bit aufweisen, ergeben sich auch neue Implementierungsbeschränkungen:

- Es darf nicht mehr als 32767 Klassen geben (aufgrund von `new`, `checkcast`).
- Die Vererbungsstufe von Klassen, welche bei 0 beginnt, darf nicht höher als 3 werden (aufgrund von `checkcast`).
- Es darf nicht mehr als 127 Methoden pro Klasse geben (aufgrund von `vcall`).
- Objektmethoden dürfen nicht mehr als 127 Parameter besitzen (wegen `vcall`).

3.2 Erweitern der Symbolliste

Die Symbolliste des Compilers ist verantwortlich für die Speicherung aller deklarierter Namen und ihrer Eigenschaften und muss dementsprechend angepasst werden, um die neuen Konzepte zu unterstützen. Der Strukturknoten, welcher für die Darstellung von Typen und insbesondere Klassen verantwortlich ist, muss nun für Vererbung und Objektmethoden um die gelb markierten Zeilen erweitert werden:

```
1  class Struct {
2      public final Kind kind;          // Kind of the structure node
3      public final Struct elemType;    // Arr: Type of the array elements
4      public Map<String, Obj> locals; // Class: Map of fields & methods
5      public final Struct superType;  // Class: Type of the super class
6      public int nFields;              // Class: Number of fields
7      public int nMethods;             // Class: Number of methods
8      public int tdAdr;                // Class: Type descriptor address
9      public int level;                // Class: Hierarchy level
10 }
```

Das Feld `superType` verweist auf den Strukturknoten der Oberklasse oder hat den Wert `null`, falls von keiner Klasse geerbt wird. In MicroJava gibt es im Gegensatz zu Java keine `Object`-Klasse, von der alle anderen Klassen implizit erben.

Als nächstes wurden die Felder `nFields` und `nMethods` eingeführt, die angeben, wie viele Felder bzw. wie viele Methoden eine Klasse besitzt. Vorher besaßen Klassen nur Felder, wodurch deren Anzahl einfach anhand der Anzahl der Elemente in `locals` (was vorher `fields` hieß) bestimmt werden konnte. Durch den Zusatz von Methoden und Vererbung wäre dies nicht mehr möglich, da Unterklassen die Anzahl der Felder der Oberklasse übernehmen, ohne diese Felder selbst in der Kollektion `locals` noch einmal zu speichern. Die Anzahl der Methoden wird ebenfalls von der Oberklasse übernommen.

Zudem existiert nun ein Feld `tdAdr`, welches angibt, an welcher Position in den globalen Daten später die Adresse für den Typdeskriptor der Klasse liegt. Jede Klasse benötigt nun also wie statische Variablen einen Platz in den globalen Daten. Die Typdeskriptoradresse ist dabei davon abhängig an welcher Position im Quellcode die Klasse definiert worden ist und bezeichnet im Wesentlichen eine vorlaufende Zahl. Das neue Feld `level` gibt an, auf welcher Vererbungsstufe sich eine Klasse befindet, beginnend bei 0.

Die alte Methode `findField` von Strukturknoten wird durch die neue Methode `findLocal` ersetzt, die nach Feldern und Methoden zusätzlich auch in Oberklassen sucht:

```
public Obj findLocal(String name) {
    Struct struct = this; Obj obj = null;

    while (obj == null && struct != null) {
        obj = struct.locals.get(name); // try to find in own class
        struct = struct.superType; // go to super class
    }
    return obj;
}
```

Eine weitere Anpassung im Strukturknoten muss vorgenommen werden, damit Unterklassen an Oberklassen zugewiesen werden können. Dafür wird in der Methode `boolean assignableTo(Struct dest)` die Alternative `this.isSubTypeOf(dest)` zusätzlichen zu den anderen Möglichkeiten eingefügt, die vergleicht, ob eine der Oberklassen mit dem Zuweisungsziel überein stimmt:

```
public boolean isSubTypeOf(Struct other) {
    if (this.kind != Kind.Class || other.kind != Kind.Class) return false;

    Struct type = this.superType;
    while (type != null && type != other)
        type = type.superType;
    return type == other;
}
```

Im Objektknoten wird die neue Art `Fld` eingefügt, welche für die Felder von Klassen anstatt von `Var` verwendet wird. Lokale Variablen von Methoden können so später von den Feldern der umschließenden Klasse unterschieden werden. Beim Erzeugen von Operanden mit `Fld`-Objektknoten wird der Operandentyp ebenfalls auf den bereits bestehenden Typ `Fld` gesetzt. Ansonsten werden `Fld`-Objektknoten wie `Var`-Objektknoten behandelt.

In der Scope Klasse des Compilers wird ein zusätzliches Feld `Struct _class` eingefügt, welches nur für Klassenscopes verwendet wird und auf den Strukturknoten dieser Klasse verweist. Dieses Feld wird für die Methode `findGlobal` verwendet, damit auch Felder

und Methoden von Oberklassen gefunden werden können. Die Felder und Methoden der eigenen Klasse werden automatisch gefunden, da sie sich noch in einem offenen Scope befinden. Elemente der Oberklasse befinden sich hingegen in keinem offenen Scope mehr und müssen daher über dieses neue Feld `_class` wie folgt gefunden werden:

```
1  public Obj findGlobal(String name) {
2      Obj res = findLocal(name); // first, try to search in own scope
3      if (res == null && _class != null) { // if class scope, search in class
4          res = _class.findLocal(name);
5      }
6      if (res == null && outer != null) { // otherwise search in outer scope
7          res = outer.findGlobal(name);
8      }
9      return res;
10 }
```

Des Weiteren wird noch die `insert` Methode von Scopes angepasst, damit die Anzahl der Variablen von Scopes auch bei der neuen Objektknotenart `Fld` und beim Einfügen von neuen Klassen hochgezählt wird. Dadurch verbrauchen Klassen einen Platz in den globalen Daten und können so ihre Typdeskriptoradresse erhalten.

Zuletzt bekommt die Hauptklasse `Tab` der Symbolliste eine neue `openScope` Methode, die für das Öffnen von Klassenscopes verantwortlich ist und daher sowohl den Strukturknoten der derzeitigen Klasse als auch eine Anzahl an bereits vorhandenen Variablen entgegen nimmt. Der Parser übergibt dabei die Anzahl der Felder der Oberklasse, falls vorhanden. Somit startet eine neue Klasse bereits mit allen Feldern der Oberklasse und neu deklarierte Felder bekommen eine korrekt fortlaufende Adresse, ohne die Felder der Oberklasse zu überdecken.

In der `insert` Methode der Symbolliste wird nun ebenfalls die neue Objektknotenart `Fld` berücksichtigt. Außerdem bekommen Klassen hier ihre Typdeskriptoradresse, die nach demselben Prinzip wie die Adressen für statische Variablen vergeben wird. Bei Methoden wird nun so wie bei Variablen der Level auf den Level des derzeitigen Scopes gesetzt, um später Objektmethoden einfach von normalen Methoden unterscheiden zu können. Hier der geänderte Auszug aus der `insert` Methode:

```
if (kind == Obj.Kind.Var || kind == Obj.Kind.Fld) {
    o.adr = curScope.nVars();
    o.level = curLevel;
} else if (kind == Obj.Kind.Type && type.kind == Struct.Kind.Class) {
    o.type.tdAdr = curScope.nVars();
} else if (kind == Obj.Kind.Meth) {
    o.level = curLevel;
}
```

Um all diese Änderungen in der Symbolliste zu verdeutlichen, folgt hier ein kleines Beispiel. Dazu wurden exemplarisch diese Vektoren als Klassen, welche voneinander erben, mit verschiedenen Felder und Methoden implementiert:

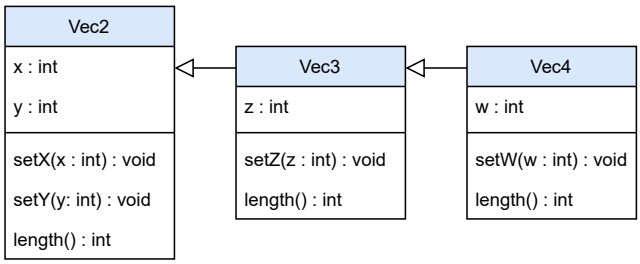


Abbildung 3.1: Klassendiagramm der Beispielklassen.

Die Symbolliste dieses Klassendiagramms wird im Folgenden schematisch dargestellt. Dabei ist zu beachten, dass alle Felder eine korrekte Adresse bekommen und so z.B. die Klasse *Vec4* auch tatsächlich 4 Felder besitzt. Außerdem wird das Feld `val` der Objektknoten nun nicht nur für Konstanten sondern auch Objektmethoden benutzt und gibt dabei eine eindeutige Methodennummer für diese Klasse an. Das heißt, dass jede Methode einer Klasse eine aufsteigende Nummer (beginnend bei 0) erhält. Unterklassen übernehmen ähnlich wie bei Felder die Anzahl der Methoden der Oberklasse und beginnen somit bei deren letzter Methodennummer zu nummerieren. Überschriebene Methoden erhalten jedoch keine eigene Methodennummer, sondern übernehmen die Nummer jener Methode, die sie überschreiben. Ebenso erhöht sich bei überschreibenden Methoden die Anzahl der Methoden einer Klasse nicht.

In Abb. 3.2 sieht man den relevanten Auszug aus der Symbolliste, in denen die Objektknoten der drei in Abb. 3.1 definierten Klassen dargestellt wurden.

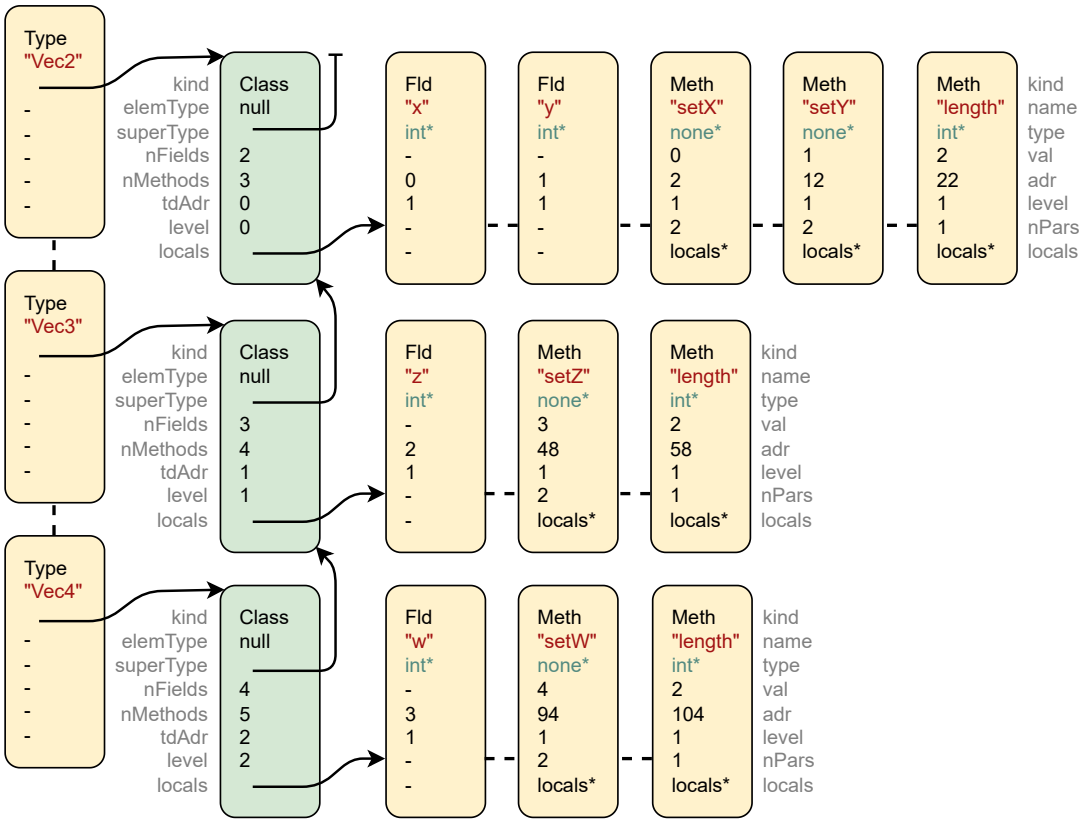


Abbildung 3.2: Schematische Übersicht der Symbolliste von Beispielklassen.

Die Verweise auf die Strukturknoten von `int` und `none` wurden dabei zwecks der Übersicht vereinfacht als Name mit Stern dargestellt. Aus gleichem Grund wurde die `locals`-Liste der Objektknoten von Methoden weggelassen. Zu beachten ist dabei jedoch, das Objektmethoden immer einen versteckten Parameter `this` an erster Stelle besitzen, wie später noch näher erläutert wird. Deswegen ist die Anzahl der Parameter `nPars` um eins höher, als man durch Abb. 3.1 zunächst vermuten würde.

Ansonsten ist in Abb. 3.2 zu erkennen, wie die Anzahl der Felder und Methoden von Unterklassen durch die Vererbung beeinflusst und wie für Felder und Methoden fortlaufende Adressen bzw. Methodennummern generiert werden. Überschriebene Methoden zählen wie angesprochen nicht doppelt und besitzen keine neue Methodennummer, allerdings haben sie andere Adressen, da sie jeweils eigene Implementierungen aufweisen. Die Typdeskriptoradresse und der Level stimmen hier nur überein, da im Programm keine anderen Klassen oder statischen Variablen vorher deklariert wurden.

3.3 Klassentabelle der Objektdatei

Die im vorherigen Kapitel besprochenen Änderungen werden zum Teil für den Parser benötigt, um die neuen semantischen Information darzustellen und zu verarbeiten. Einige der nun zusätzlich gespeicherten Informationen sind aber auch für den Interpreter wichtig, um virtuelle Methodenaufrufe und Typtests durchführen zu können. Da man jedoch nur einen kleinen Teil der Daten der Symbolliste benötigt, werden diese in vereinfachter Form in die vom Compiler generierte .obj-Datei geschrieben. Konkret speichert man für jede Klasse die Adressen all ihrer Methoden und eine Liste aller Typdeskriptoradressen ihrer Oberklassen. Die Adressen der Methoden müssen dabei anhand ihrer Methodennummer sortiert werden und auch die Typdeskriptoradressen der Oberklassen sind in aufsteigender Reihenfolge beginnend bei der Basisklasse gespeichert.

Um diese neuen Informationen in der binären .obj-Datei darzustellen, ist auch die Anzahl der Klassen und die jeweiligen Anzahlen von Methoden und Oberklassen mitzuspeichern. Die angepasste Grammatik der Objektdatei sieht dadurch wie folgt aus:

```
ObjFile =  "MJ" codeSize dataSize mainPC
           nClasses {nLevels {tdAdr} nMethods {methodAdr}}
           Code.
   Code =  {byte}.
```

Dabei ist zu beachten, dass die Liste der Typdeskriptoradressen von Oberklassen an der letzten Position immer die Typdeskriptoradresse der eigenen Klasse enthält, diese wird später im Interpreter gesondert benötigt und ist an dieser Position auch für den Typtest wichtig. Um das ganze etwas anschaulicher zu gestalten, sieht man die Klassentabelle für das Beispiel aus Abb. 3.2 hier in Tabellenform:

nClasses	3						
nLevels	1	**tdAdr:**	0				
nMethods	3	**methodAdr:**	2	12	22		
nLevels	2	**tdAdr:**	0	1			
nMethods	4	**methodAdr:**	2	12	58	48	
nLevels	3	**tdAdr:**	0	1	2		
nMethods	5	**methodAdr:**	2	12	104	48	94

Tabelle 3.1: Beispielhafte Klassentabelle in einer Objektdatei.

Diese Klassentabelle der Objektdatei wird nun im Interpreter geladen und für die spätere Code-Ausführung vorbereitet. Dabei wird für jede Klasse jeweils Platz am Heap für die Basistypliste und die Methodenliste gemeinsam reserviert. In den globalen Daten wird die Adresse des erzeugten Typdeskriptors im Heap an die Position der Typdeskriptoradresse der Klasse gespeichert. Hierbei erkennt man den Unterschied zwischen Typdeskriptoradresse und Typdeskriptor: die Typdeskriptoradresse ist die Position des Typdeskriptors in den globalen Daten, der Typdeskriptor ist ein Verweis auf einen Speicherbereich für Basistyp- und Methodenliste am Heap.

Die ersten vier Einträge des Typdeskriptors stellen die Basistypliste dar, der Rest sind die Adressen der Methoden im Code. Die Sortierung dieser Listen wird dabei von der Objektdatei übernommen. Enthält die Basistypliste weniger als vier Einträge, so wird der Rest mit 0 aufgefüllt. Diese leeren Einträge sind später für die Ausführung des Typtests wichtig. Die fixe Anzahl von vier Einträgen wurde dabei gewählt, um nicht zu viel Platz am Heap zu verbrauchen und dennoch brauchbare Vererbungstiefen im Code zu erlauben. Dieses Limit könnte problemlos auf bis zu 127 erhöht werden, danach wäre der 8 Bit große Parameter des Typtest-Bytecodes der limitierende Faktor.

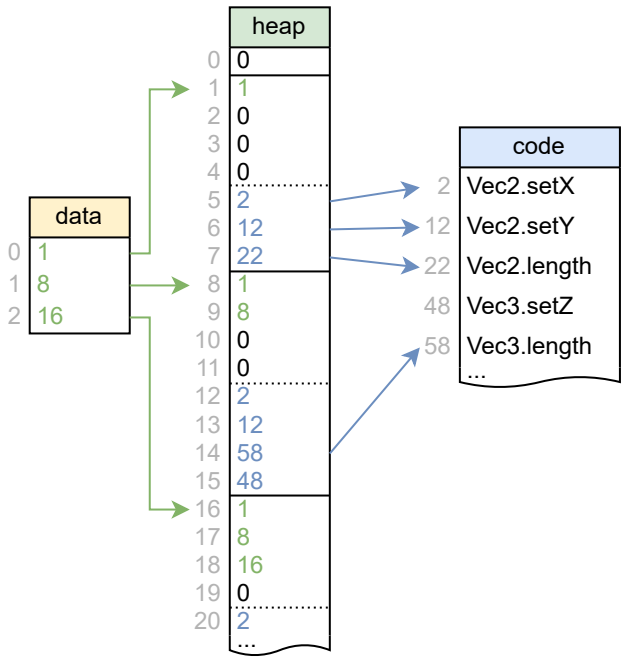


Abbildung 3.3: Darstellung der geladenen Klassentabelle im Interpreter.

In Abb. 3.3 sieht man den Zustand des Interpreters nach dem Laden der Objektdatei aus dem vorherigen Beispiel. Das Codesegment wurde dabei vereinfacht dargestellt und einige Pfeile von den Methodenlisten auf das Codesegment wurden zwecks der Übersicht ausgelassen. Es lässt sich trotzdem gut erkennen, wie Methoden- und Basistyplisten abgespeichert werden. Der erste Platz im Heap ist übrigens für die Adresse von `null` reserviert und darf nicht anderweitig verwendet werden.

3.4 Anpassung des Parsers

Da der Parser gewissermaßen die Steuereinheit des Compilers ist, sind hier einige Änderungen notwendig. Das Verarbeiten der neuen Syntax kann dabei jedoch sehr systematisch programmiert werden und bringt keine neuen Konzepte gegenüber dem bestehenden Compiler, weshalb hier nicht näher darauf eingegangen wird. Die Generation der Bytecodes wird hingegen in eigenen Kapiteln beschrieben. Trotzdem sind noch einige weitere Anpassungen erforderlich, die hier näher erläutert werden.

Zuerst erfolgt eine kleine Anpassung der Methode `VarDecl`, welche vorher immer implizit Objektknoten vom Typ `Var` eingefügt hat, doch nun einen neuen Parameter bekommt, der die Art der eingefügten Objekte angibt. So kann für Variablendeklarationen in Klassen die Art direkt auf den neuen Typ `Fld` gesetzt werden.

Anschließend wird die Methode `ClassDecl` verändert. Sollte die Klasse optional von einer anderen erben, so muss die Kontextbedingung überprüft werden, dass es sich tatsächlich um eine weitere Klasse handelt. Da dies ein Einpass-Compiler ist, und eine neue Klasse erst nach dem Verarbeiten der optionalen Vererbung in die Symbolliste eingefügt wird, sind zyklische Vererbungsabhängigkeiten automatisch nicht möglich, da jeweils eine Klasse noch nicht gefunden werden kann. Aus demselben Grund sind in MicroJava auch keine indirekten Rekursionen möglich.

Sollte aber von einer Klasse geerbt werden, so werden beim Erzeugen des neuen Strukturknoten die Felder `supertype` und `level` anhand der Oberklasse korrekt gesetzt. An dieser Stelle wird ebenfalls gleich überprüft, ob das Limit des Vererbungslevels von 3 nicht überschritten wird. Beim Öffnen des Klassenscopes wird anschließend die vorher beschriebene neue `openScope` Methode benutzt:

```
tab.openScope(superType == null ? 0 : superType.nFields, clazz.type);
```

Danach wird beim Einlesen von Variablen- und Methodendeklarationen in `ClassDecl` die vorher bereits beschriebene Methode `isMethod` verwendet, um zu entscheiden, welche Alternative betreten werden soll:

```
while (sym == ident || sym == void_) {
    if (isMethod()) {
        MethodDecl(clazz.type);
    } else {
        VarDecl(Obj.Kind.Fld);
    }
}
```

Abschließend wird in `ClassDecl` noch überprüft, ob die Anzahl der Methoden in der Klasse das Limit von 127 nicht überschreitet.

Als nächstes bekommt die Methode `MethodDecl` einen neuen Parameter `Struct clazz`, der für Objektmethoden auf den Strukturknoten der Klasse verweist. Ist dieser gesetzt, so wird der Parameter `this` mit dem Typ der aktuellen Klasse eingefügt. Des Weiteren wird anhand des Methodennamens überprüft, ob es sich um eine überschreibende Methode handelt, wodurch die Methodennummer dementsprechend festgelegt wird:

```
if (clazz != null) {
    tab.insert(Obj.Kind.Var, "this", clazz); // insert "this" parameter

    if (clazz.superType != null) {}
        overrideMeth = clazz.superType.findLocal(curMethod.name);
    }
    if (overrideMeth != null && overrideMeth.kind == Obj.Kind.Meth) {
        curMethod.val = overrideMeth.val; // use same method number
    } else { // use new method number
        curMethod.val = clazz.nMethods; clazz.nMethods++;
    }
}
```

Sollte es sich um eine überschreibende Methode handeln, so wird nach dem Einlesen der Parameterliste überprüft, ob die Parameter und der Rückgabewert mit der überschriebenen Methode überein stimmen, ansonsten wird ein Fehler gemeldet.

In der Methode `ActPars` muss dann der versteckte Parameter `this` von Objektmethoden berücksichtigt werden, indem man ihn beim Vergleichen von formellen zu tatsächlichen Parametern überspringt.

Um Typumwandlungen zu ermöglichen, muss in der Methode `Factor` die richtige Alternative anhand von semantischen Informationen ausgewählt werden, was diesmal ohne Vorausschauen von Symbolen möglich ist. Bezeichnet das *ident* Symbol nach einer geöffneten Klammer einen Typ, so wird von einem Cast ausgegangen, ansonsten von einem geklammerten Ausdruck. Bei einem Cast wird der Typ des Objektknotens geändert:

```
case lpar:
    scan();
    if (sym == ident && tab.find(la.str).kind == Obj.Kind.Type) { // cast
        Struct castType = Type();
        check(rpar);
        x = Expr();
        // TODO: code generation for cast...
        x.type = castType;
    } else { // expression in parenthesis
        x = Expr();
        check(rpar);
    }
```

Zuletzt wird noch die Methode `Designator` angeglichen. Die Produktion *Designator* hat zur Erinnerung folgende Grammatik:

`Designator = ident`$_0$ `{"." ident`$_1$ `| "[" Expr "]"}.`

Für das Symbol $ident_0$ kann es nun vorkommen, dass es sich bereits um ein Feld oder eine Objektmethode handelt. Da Felder und Objektmethoden nur in Klassenscopes eingefügt und durch die `find` Methode der Symbolliste nur von **offenen** Scopes gefunden werden können, muss es sich um Felder oder Objektmethoden der **eigenen** Klasse handeln. Dies wiederrum bedeutet, dass gerade implizit der `this` Parameter verwendet wurde, der deshalb von der fixen Position 0 der lokalen Variablen geladen werden muss:

```
check(ident);
Operand x = new Operand(tab.find(t.str), this);

if (x.kind == Operand.Kind.Fld ||
        (x.kind == Operand.Kind.Meth && x.obj.level > 0)) {
    code.put(OpCode.load_0); // load implicit "this" parameter
}
```

Zusätzlich ist zu beachten, dass $ident_1$ eines *Designators* nun nicht nur ein Feld, sondern auch eine Methode bezeichnen kann und daher der zurückgegebene Operand jeweils korrekt gesetzt werden muss.

3.5 Codeerzeugung für Objektmethoden

Die Anpassungen im vorherigen Kapitel erlauben es, Objektmethoden bereits fast wie die existierenden, statischen Methoden zu behandeln, da sie durch das implizite Einfügen und Laden des `this` Parameters wie gewöhnlich funktionieren. Die Objektmethoden rechts in folgenden Beispiel werden also intern wie die statischen Methoden links im Beispiel behandelt:

```
class A {
    int a;
    void set(int v) { a = v; }
    int get() { return a; }
}
```

```
class A {
    int a;
}
void set(A this, int v) { this.a = v; }
int get(A this) { return this.a; }
```

Allerdings bereitet das Überschreiben von Methoden ein Problem, welches sich nicht ganz so einfach lösen lässt, da zur Kompilationszeit nicht klar ist, welche Methode eigentlich aufgerufen wird. Dies hängt rein vom dynamischen Typ des Objekts ab. Der dynamische Typ ist bisher aber selbst zur Laufzeit nicht bekannt, da Objekte rein auf die Daten ihrer Felder verweisen und keine zusätzlichen Informationen abspeichern. Um dies zu ändern, bekommt der Bytecode `new` einen zusätzlichen Parameter, welcher die Typdeskriptoradresse und somit den dynamischen Typ angibt. Hier ein Beispiel mit den Klassen aus Abb. 3.1:

```
Vec2 a = new Vec2; // generates bytecode: new 3, 0
Vec2 b = new Vec3; // generates bytecode: new 4, 1
Vec2 c = new Vec4; // generates bytecode: new 5, 2
```

Der erste Parameter des Bytecodes `new` gibt dabei wiederrum die Größe des Objekts an, muss aber immer um 1 höher als vorher gesetzt werden, da die Typdeskriptoradresse und damit der neue Marker des Objekts ebenfalls Platz benötigt. Im Parser erfolgt die Codegenerierung für die neue Objekterzeugung also wie folgt:

```
code.put(OpCode.new_);
code.put2(type.nFields + 1); // +1 for the type tag
code.put2(type.tdAdr);
```

Um die virtuellen Methodenaufrufe zur Laufzeit durchführen zu können, braucht es einen neuen Bytecode `vcall`, welcher zwei Parameter aufweist: die Methodennummer und die Anzahl der Parameter, einschließlich dem versteckten Parameter `this`. Im Codegenerator des Compilers wird nun beim Generieren von Methodenaufrufen überprüft, ob es sich um eine Objektmethode handelt, wodurch dann `vcall` anstatt `call` verwendet wird:

```
public void methodCall(Operand x) {
    ...
    } else if (x.obj.level > 0) { // --- object method call
        put(OpCode.vcall);
        put(x.obj.val); // method number
        put(x.obj.nPars);
    } else { // --- normal method call
        put(OpCode.call);
        put2(x.adr - (pc - 1)); // method addresse
    }
}
```

3.6 Virtuelle Methodenaufrufe im Interpreter

Jetzt wo der Compiler für Objektmethoden fertig ist und bereits neuen Bytecode erzeugt, muss auch der Interpreter angepasst werden, um die neuen Instruktionen ausführen zu können. Zuerst wird die Implementierung von `new` angepasst:

```
case new_:
    int newAdr = alloc(next2() * 4); // reserve size in heap
    heap[newAdr] = data[next2()]; // put type tag
    push(newAdr + 1); // return object address
    break;
```

Die Methode `next2` holt sich dabei jeweils die nächsten zwei Byte des Bytecodes und somit jeweils einen der Parameter von `new`. Zuerst wird wie vorher ein Bereich am Heap mit der Methode `alloc` des Interpreters reserviert. An die erste Position dieses reservierten Bereichs wird nun der Typdeskriptor von den globalen Daten anhand der Typdeskriptoradresse geladen. Als Objektadresse wird anschließend die zweite Position des Bereichs zurückgegeben, wodurch die Adressen von Feldern des Objekts weiterhin stimmen.

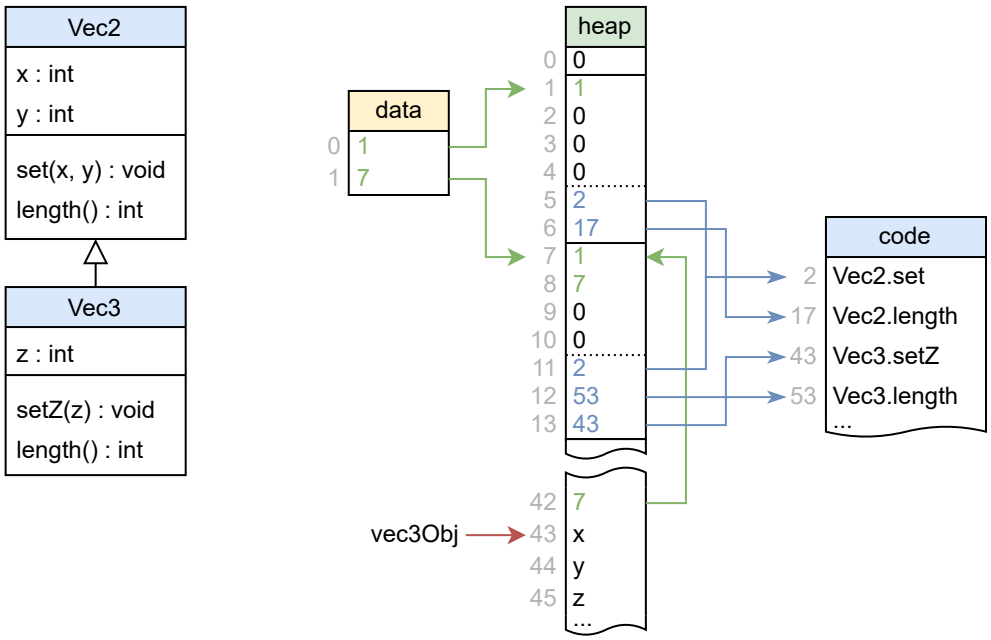


Abbildung 3.4: Darstellung eines Objekts mit Typdeskriptor-Marker im Interpreter.

In Abb. 3.4 ist ein erzeugtes Objekt im Interpreter abgebildet. Wie man erkennt, befindet sich nun immer eine Position vor der Objektadresse ein Verweis auf den Typdeskriptor.

Die Implementierung der `vcall` Instruktion benutzt all diese vorbereiteten Daten, um virtuelle Methodenaufrufe sehr effizient durchzuführen. Die Methodennummer und Anzahl der Methodenparameter bekommt man aus den Parametern der Bytecode Instruktion. Die ebenfalls benötigte Objektadresse kann anhand der Methodenparameter vom Expressionstack geladen werden, da als erster Parameter einer Objektmethode immer das Objekt selbst übergeben wird. Durch die Objektadresse gelangt man zum Typdeskriptor, wodurch man gemeinsam mit der Methodennummer die gewünschte Methodenadresse erhält, zu jener anschließend gesprungen wird.

```
1  case vcall:
2      int mNum = next();
3      int nPars = next();
4      adr = stack[esp - nPars]; // get object address
5      if (adr == 0) {
6          throw new IllegalStateException("null reference used");
7      }
8      int tab = heap[adr - 1];
9      PUSH(pc);
10     pc = heap[tab + SUPER_TABLE_LENGTH + mNum]; // skip supertype table
11     break;
```

Wie aus der Implementierung zu erkennen ist, benötigt man noch einen Sicherheitscheck. Denn wenn die Objektadresse gleich 0 ist, bezeichnet sie `null` und hat somit keinen Typ, wodurch der Versuch einen Typdeskriptor zu bekommen scheitern würde. In Java würde hier eine `NullPointerException` geworfen werden[4], da Ausnahmebehandlung erst später eingeführt wird und man dann diese Klasse auch vordefinieren müsste, wird hier aus Gründen der Einfachheit der Interpreter mit einer passenden Nachricht beendet.

Darüber hinaus ist das Ablegen des Programmzeigers (program counter - pc) auf den Methodenstack, wie bei einem gewöhnlichen Aufruf, zu beachten. Außerdem ist beim Nachschlagen der Methodenadresse die Methodennummer um die Länge der Basistypliste zu erhöhen, damit diese übersprungen wird, wie man auch in Abb. 3.4 erkennen kann. Eine Methode liegt also auf Adresse Typdeskriptor + Methodennummer + Basistyplistenlänge, was beispielsweise für `setZ` gleich 7 + 2 + 4 = 13 entspricht.

3.7 Codeerzeugung für Typtest und Cast

Um einen Typtest durchzuführen, benötigt es einen neuen Bytecode namens `checkcast`, welcher als Parameter die Typdeskriptoradresse mit 16 Bit und den Level des Überprüfungstyps mit 8 Bit entgegen nimmt. Zusätzlich muss dabei die Adresse des zu überprüfenden Objekts am Expressionstack liegen. Diese Objektadresse wird durch die Instruktion entfernt und nach der Ausführung von `checkcast` liegt am Expressionstack stattdessen eine 1, falls das Objekt vom Typ des Überprüfungstyps war, ansonsten eine 0.

Da man anhand dieses Ergebnisses zu unterschiedlichen Codeadressen springen muss, werden zwei weitere Bytecode Instruktionen eingeführt: `jt` ('jump true') und `jf` ('jump false'). Alle bisherigen Sprunginstruktionen von MicroJava verlangen entweder zwei Parameter zum Vergleichen oder sind unbedingt. Die Instruktion `jf` springt dabei zur mitgegeben Adresse, falls genau eine 0 am Expressionstack liegt, `jt` springt bei allem außer einer 0. Dabei wurden trotz der Ähnlichkeit beide Varianten (`jt` & `jf`) implementiert, damit sie sich später nahtlos in die Kurzschlussauswertung von Bedingungen einfügen.

Um nun den Code für Typtests und Casts erzeugen zu können, wird im Codegenerator eine neue Methode eingefügt:

```
1  public void checkCast(Operand x, Struct checkType) {
2      if (checkType.kind != Struct.Kind.Class) {
3          parser.error(CAST_NOT_CLASS);
4      } else if (x.type.kind != Struct.Kind.Class) {
5          parser.error(CAST_NOT_OBJECT);
6      } else if (x.type != checkType &&
7              !x.type.isSubTypeOf(checkType) &&
8              !checkType.isSubTypeOf(x.type)) {
9          parser.error(CANNOT_CAST);
10     }
11     put(OpCode.checkcast); put2(checkType.tdAdr); put(checkType.level);
12 }
```

Dabei werden zuerst die drei Kontextbedingungen für Typtests geprüft. Sowohl das zu überprüfende Objekt als auch der Überprüfungstyp müssen eine Klasse bezeichnen und beide Klassen müssen miteinander verwandt sein. Auch bei Java gibt es bei unverwandten Klassen bereits zur Kompilationszeit einen Fehler.[3]

Um nun Typumwandlung in MicroJava zu implementieren, muss auch überlegt werden, was im Falle eines nicht durchführbaren Casts geschehen sollte. In Java würde eine `ClassCastException` geworfen werden[5], doch wie vorher bei Methodenaufrufen, wird hier ein einfacherer Weg gewählt. In MicroJava existiert bereits ein `trap` Bytecode, welcher das Programm mit einem Fehlercode beendet. Für fehlgeschlagene Casts wird nun der neue Fehlercode '2' eingeführt. Der Bytecode jeder Objektdatei beginnt mit solch einer `trap 2` Anweisung, zu der im Fehlerfall gesprungen werden kann.

```
1 code.load(x);
2 code.put(OpCode.dup);
3 code.checkCast(x, castType);
4 code.put(OpCode.jf);
5 castTrap.put();
```

Das obige Codestück wird im bereits gezeigten Code von Kapitel 3.4 an der Stelle des `TODO`-Kommentars eingefügt. Die Objektadresse muss dabei mithilfe von `dup` am Expressionstack dupliziert werden, damit sie später an die jeweilige Variable zugewiesen werden kann. Im Fehlerfall wird zu dem vorher besprochenen Label `castTrap` gesprungen und damit das Programm beendet.

In der Klasse `CompOp`, die für Vergleichsoperationen zuständig ist, werden die neuen besprochenen Arten `jt` und `jf` eingefügt. So kann in der Methode `CondFact` des Parsers der Typtest sehr einfach hinzugefügt werden, womit er auch automatisch korrekt in der Kurzschlussauswertung behandelt wird.

```
1 if (sym == instanceof_) {
2     scan();
3     Struct checkType = Type();
4     code.checkCast(x, checkType);
5     x = new Operand(CompOp.t, code);
6 } else {
7     // code for relational conditions like before...
8 }
```

3.8 Typtest im Interpreter

Der Vollständigkeit wegen seien zuerst die Implementierungen für `jt` und `jf` gegeben, welche sehr ähnlich zu den restlichen Sprüngen sind:

```
case jt:
  off = next2();
  if (pop() != 0) pc += off - 3; // -3 because of instruction length
  break;
case jf:
  off = next2();
  if (pop() == 0) pc += off - 3;
  break;
```

Interessanter ist hingegen die Implementierung von `checkcast`. Zuerst werden dabei die zwei Parameter, Level und Typdeskriptoradresse des Überprüfungstyps, eingelesen. Diese Typdeskriptoradresse wird dabei anhand der globalen Daten gleich in den Typdeskriptor umgewandelt. Die Adresse des zu überprüfenden Objekts liegt am Expressionstack und kann von dort geholt werden. Sollte das Objekt gleich `null` sein, so wird sofort eine 1 zurückgegeben, da ein Cast mit `null` immer funktioniert. Da in MicroJava `checkcast` auch für den Typtest verwendet wird, ist das Ergebnis von `null instanceof T` also immer `true`. Dies unterscheidet sich zu Java, wo es für den Typtest unter anderem aus dem Grund eine eigene Instruktion gibt, damit bei der Überprüfung von `null` das Ergebnis `false` geliefert werden kann.[6]

Ist das Objekt nicht `null`, so erfolgt der eigentliche Typtest, der anhand der vorbereiteten Basistyplisten wieder sehr effizient erfolgen kann. Auf den Typdeskriptor des zu überprüfenden Objekts kann wieder anhand des Markers vor der Objektadresse zugegriffen werden, wodurch man sofort zur Basistypliste an den ersten 4 Positionen des Typdeskriptors gelangt. Nun sieht man in dieser Liste an der Position des Levels nach. Sollte sich dort derselbe Typdeskriptor wie der des Objekt befinden, so ist das Objekt vom Überprüfungstyp und es wird eine 1 zurückgegeben, ansonsten eine 0.

```
case checkcast:
    int td = data[next2()];
    int level = next();
    adr = pop(); // object address
    if (adr == 0) {
        push(1); // null cast always works
    } else {
        int tag = heap[adr - 1];
        push(heap[tag + level] == tdAdr ? 1 : 0);
    }
    break;
```

Dies funktioniert, da der Eintrag der Basistypliste an der Stelle des Levels nur mit dem Typdeskriptor des Überprüfungstyps überein stimmen kann, wenn das Objekt auch tatsächlich diesen Typdeskriptor als Basistyp hat und damit vom Überprüfungstyp abgeleitet ist. Leere Einträge der Basistypliste haben den Wert 0, wodurch bei einem zu hohen Level (z.B. im Fall `new Vec2 instanceof Vec3`) der Typdeskriptor niemals unabsichtlich übereinstimmen kann, da 0 eine reservierte Adresse ist und nicht für Typdeskriptoren verwendet wird.

Zum Abschluss von Kapitel 3 hier noch eine Übersicht über alle geänderten oder neuen Bytecode Instruktionen:

Name	Parameter 1		Parameter 2		Expressionstack	
	Beschreibung	Bit	Beschreibung	Bit	davor	danach
vcall	Methodennr.	8	Anz. Parameter	8	..., params	..., params
new	Objektgröße	16	Deskriptoradr.	16	...	..., adr
jt	Sprungdistanz	16	-		..., 0 | 1	...
jf	Sprungdistanz	16	-		..., 0 | 1	...
checkcast	Deskriptoradr.	16	Level	8	..., obj	..., 0 | 1

Tabelle 3.2: Tabelle neuer und geänderter Bytecode Instruktionen.

4 Ausnahmebehandlung

Nach Objektorientierung soll MicroJava auch um Ausnahmebehandlung erweitert werden, ein Konzept welches zurzeit noch nicht vorhanden ist. Dabei wird die neue Syntax wieder sehr ähnlich zu Java gehalten[7], die Funktionalität aber etwas einfacher. So gibt es zum Beispiel keine checked exceptions und es können beliebige Klassen geworfen werden. Für die Implementierung der Ausnahmebehandlung sind teils wieder ähnliche Schritte wie bei der Objektorientierung notwendig. Wiederum muss die Grammatik der Sprache und die Informationen in der Objektdatei erweitert werden. Außerdem gibt es ebenfalls wieder neue Bytecode Instruktionen, wodurch Parser und Interpreter angepasst werden müssen. Generell ist der Aufwand aber geringer als vorher, die Symbolliste muss hier zum Beispiel nicht mehr verändert werden.

4.1 Syntax und Kontextbedingungen

Um Ausnahmebehandlung in der Syntax von MicroJava zu unterstützen, muss also abermals die Grammatik erweitert werden. Es existieren nun drei neue Schlüsselwörter, die im Scanner und als Tokentyp definiert werden müssen: `try`, `catch` und `throw`.

Zuerst wird eine neue Produktion namens *Catch* eingefügt, die einen catch-Block darstellt. Zu beachten gilt, dass hier keine neue Variable deklariert, sondern nur eine bereits bestehende Variable gefangen werden kann, da in MicroJava alle Variablen einer Methode am Anfang deklariert werden müssen.

```
Catch = "catch" "(" ident ")" Block.
```

Kontextbedingungen:

- *ident* muss eine Variable mit dem Typ einer Klasse sein.

Als nächstes werden zwei neue Alternativen in der Produktion *Statement* eingefügt, jeweils für einen try-catch-Block und `throw`. Der Rest der *Statement* Produktion ändert sich nicht und wurde hier zwecks der Übersicht ausgelassen:

```
Statement = ... | "try" Block Catch {Catch} | "throw" Expr ";" | ...
```

Kontextbedingungen:

- Der Typ von *Expr* muss eine Klasse sein.

4.2 Codeerzeugung für try-catch und throw

Um einen korrekten Codefluss für try-catch-Blöcke zu garantieren, muss am Ende des try-Blocks und am Ende jedes catch-Blocks ein Sprung zu dem Code nach try-catch eingefügt werden, wie in folgendem Beispiel zu sehen ist:

```
try {
    n = getNext(); // can throw E1
    if (n == 0) {
        throw new E2;
    }
    res = m / n;
} catch(e1) {
    err = -1;
} catch(e2) {
    err = -2;
}
if (err == 0) {
    ...
}
```

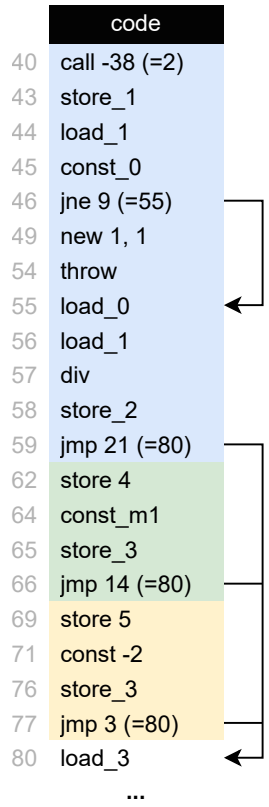


Abbildung 4.1: Codeerzeugung für einfaches try-catch Beispiel.

Die Codeerzeugung für das Werfen von Ausnahmen mithilfe von `throw` ist im Parser recht einfach. Nach dem Überprüfen der Kontextbedingung wird das zu werfende Objekt geladen und der Bytecode `throw` generiert:

```
1 Operand e = Expr();
2 if (e.type.kind != Struct.Kind.Class) error(THROW_NOT_CLASS);
3 code.load(e);
4 code.put(Code.OpCode.throw_);
```

4.3 Generieren der Ausnahmetabellen

Damit der Interpreter den Befehl `throw` ausführen kann, benötigt er Informationen, in welchen Codebereichen welche Ausnahmen gefangen werden und wohin im Ausnahmefall gesprungen wird. Dazu muss wiederum die Grammatik der Objektdatei erweitert werden, wobei aufgrund des binären Formates auch die Anzahl der Einträge in der Tabelle übergeben wird:

```
ObjFile =  "MJ" codeSize dataSize mainPC
           nClasses {nLevels {tdAdr} nMethods {methodAdr}}
           nCatches {fromAdr toAdr tdAdr catchAdr}
           Code.
   Code =  {byte}.
```

Um diese Ausnahmetabelle zu generieren, merkt sich der Parser den Programmzeiger am Anfang und Ende des try-Blocks, sowie am Anfang jedes catch-Blocks. Diese Werte werden gemeinsam mit der Typdeskriptoradresse der gefangenen Klasse in eine Liste eingefügt. Für das Beispielprogramm aus Abb. 4.1 würde die Tabelle wie folgt aussehen:

nCatches	3		
from	**to**	**tdAdr**	**catchAdr**
40	59	0	62
40	59	1	69

Tabelle 4.1: Beispielhafte Ausnahmetabelle in einer Objektdatei.

Dabei ergibt sich durch die Einfüge-Reihenfolge im Parser auch automatisch die richtige Sortierung dieser Liste für die spätere Suche nach dem passenden catch-Block. Im Code früher geschriebene catch-Blöcke werden zuerst geprüft und bei geschachtelten try-catch Konstrukten wird von innen nach außen gearbeitet.

4.4 Ausnahmebehandlung im Interpreter

Um die `throw`-Anweisung implementieren zu können, müssen zunächst die Anweisungen für den Auf- und Abbau von Methodenframes, `enter` und `exit`, leicht angepasst werden. Dabei wird nun zusätzlich zum Framepointer auch der aktuelle Expressionstackpointer auf dem Methodenstack in `enter` abgelegt und in `exit` wieder abgebaut, damit man in `throw` auch den Expressionstack kontrolliert abbauen kann, ohne noch wichtige Werte von früheren Methoden zu zerstören.

Nun zur Implementierung der `throw`-Anweisung: Diese holt sich zunächst das Ausnahmeobjekt vom Expressionstack. Im Fall von `null` gibt es eine Fehlermeldung, ansonsten werden alle Methoden am Methodenstack nacheinander abgebaut, wobei jeweils mit der Hilfsmethode `getHandler` nach einer Adresse eines passenden catch-Blocks gesucht wird.

```
1  case throw_:
2      int e = pop();
3      if (e == 0) throw new IllegalStateException("null thrown");
4      adr = pc; // throw adr
5      for(;;) { // for all methods on stack
6          int h = getHandler(adr, e);
7          if (h >= 0) { pc = h; break; }
8          sp = fp;
9          POP(); // esp
10         fp = POP();
11         if (fp == 0) throw new IllegalStateException("no catch");
12         esp = local[fp - 1];
13         adr = POP(); // return adr
14     }
15     push(e);
16     break;
```

Sollte diese Hilfsmethode fündig geworden sein (Rückgabewert >= 0), so setzt man den Programmzeiger auf die gefundene Adresse, gibt das Ausnahmeobjekt wieder auf den Expressionstack zurück und das Programm wird in diesem catch-Block normal fortgeführt. Sollte keine passende Fangadresse gefunden worden sein, so wird der aktuelle Methodenframe abgebaut und als neue Adresse der geworfenen Ausnahme wird die Rückkehradresse der Methode benutzt, wie in Abb. 4.2 zu erkennen ist.

Damit kann der Prozess für die nächste Methode von vorne beginnen, es sei denn, der neue Framepointer ist 0, was bedeutet, dass keine Methode mehr auf dem Methodenstack liegt. Dies kann passieren, wenn eine geworfene Ausnahme nirgends gefangen wird und man beim Abbauen des Methodenstacks gerade die `main`-Methode behandelt hat. In diesem Fall kann das Programm nur noch mit einer Fehlermeldung beendet werden. Zusätzlich kann aufgrund der modifizierten `enter` und `exit` Anweisungen auch der Expressionstack korrekt abgebaut werden.

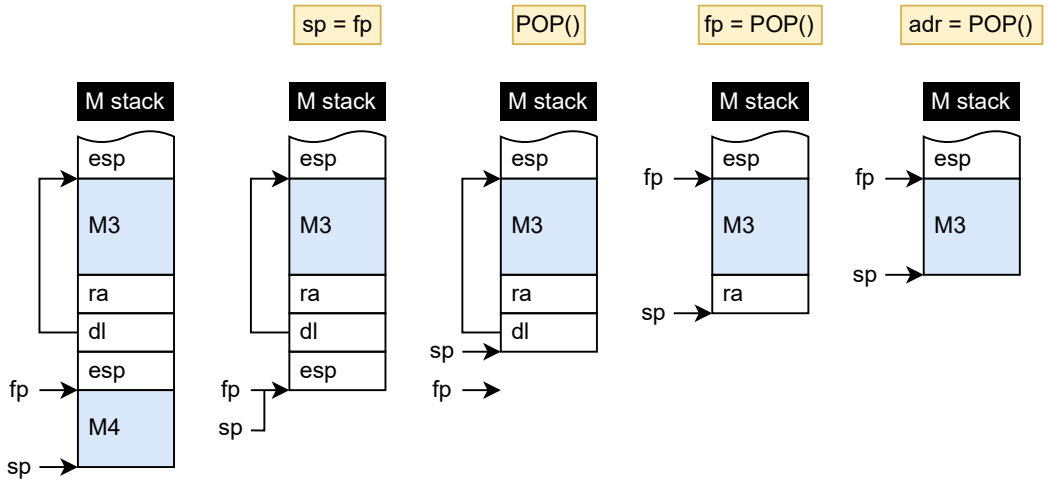


Abbildung 4.2: Abbau des Methodenstacks in der `throw`-Anweisung.

In Abb. 4.2 ist zu erkennen, wie Stackpointer und Framepointer sich während des Methodenabbaus verhalten. Sollte eine vorherige Methode existieren, so kann man durch `esp = local[fp-1]` vor dem letzten Schritt den Expressionstackpointer korrekt für die nächste Methode leeren. Danach wird noch die Rückkehradresse als nächste Adresse für das Suchen der catch-Blöcke gesetzt.

Für die `throw`-Anweisung wird also die Hilfsmethode `getHandler` benötigt, welche für das Finden der richtigen Fangadresse eines geworfenen Objekts anhand dessen Wurfadresse im Code verantwortlich ist. Dafür wird die Ausnahmetabelle Zeile für Zeile durchgegangen und zunächst überprüft, ob sich die Wurfadresse im Bereich eines Eintrags der Tabelle befindet. Danach muss kontrolliert werden, ob dieser Eintrag und damit auch der catch-Block tatsächlich für das geworfene Objekt verantwortlich ist. Dazu benutzt man die Basistypliste des Objekts und vergleicht, ob der Typdeskriptor eines Basistyps mit dem Typdeskriptor des Tabelleneintrags übereinstimmt, in welchem Fall die dazugehörige Fangadresse zurückgeben wird. An der letzten Position enthält die Basistypliste immer den eigenen Typdeskriptor, wodurch man das Ende der Liste erkennen kann. Sollte keiner der Typdeskriptoren der Liste zum derzeitigen catch-Block passen, geht man in der Ausnahmetabelle zum nächsten Eintrag über. Wenn keiner der Einträge passt, wird -1 zurückgegeben.

```
1  private int getHandler(int throwAdr, int e) {
2      int length = heap[exceptionTab];
3      int tag = heap[e - 1]; // get tag from exception object
4
5      for (int i = exceptionTab + 1; i <= exceptionTab + length; i += 4) {
6          int fromAdr = heap[i];
7          int toAdr = heap[i + 1];
8          int catchTd = heap[i + 2];
9          int catchAdr = heap[i + 3];
10         // --- check if throwAdr in range
11         if (fromAdr > throwAdr || toAdr < throwAdr) continue;
12         // --- check all supertypes
13         for(int superType = tag; ; superType++) {
14             if (heap[superType] == catchTd)
15                 return catchAdr; // return catch address
16             else if (heap[superType] == tag)
17                 break; // end of supertype table reached
18         }
19     }
20     return -1; // no catch found for throwAdr
21 }
```

5 Erstellen von Testfällen

Um das Vertrauen in die korrekte Funktionsweise von Compiler und Interpreter zu erhöhen, ist ausführliches und systematisches Testen notwendig. MicroJava hatte bereits ein Test-Framework auf Basis von JUnit, welches in diesem Kapitel für die neuen Funktionen der Sprache erweitert wird. Das Framework funktioniert, in dem es zunächst mit einem MicroJava-Programm als String initialisiert wird. Für einen besseren Überblick wird dieser String üblicherweise direkt in den Test-Methoden definiert. Danach kann man dem Framework mitteilen, dass man sich einen bestimmten Fehler oder eine bestimmte Befehlszeilenausgabe erwartet. Das Framework kompiliert nun das Programm und vergleicht eventuelle Fehlerausgaben des Compilers mit den erwarteten. Sollte das Programm korrekt sein, wird es ausgeführt und die Ausgabe mit der erwarteten verglichen. Sollten die Ergebnisse nicht übereinstimmen, schlägt der Unit-Test fehl.

Zunächst wurde die Einhaltung und die korrekte Fehlerausgaben des Compilers für die Kontextbedingungen überprüft. Der Test, dass Klassen nicht von sich selbst erben dürfen sieht dabei zum Beispiel so aus:

```
1  @Test
2  public void noSelfExtend() {
3      init("program P\n" +
4          "  class A extends A { int a; }\n" +
5          "{\n" +
6          "  void main() {}\n" +
7          "}");
8      expectError(2, 21, NOT_FOUND, "A");
9      parseAndVerify();
10 }
```

Die Methode `expectError` teilt hier dem Test-Framework mit, dass der Fehler namens `NOT_FOUND` mit Argument 'A' in Zeile 2 - Spalte 21 erwartet wird. Nach ähnlichem Prinzip

gibt es für das Erwarten von Befehlszeilenausgaben die Methode `addExpectedRun` des Frameworks. Nachdem man alle Vorbereitungen getroffen hat, ruft man die Methode `parseAndVerify` auf, die dann den eigentlichen Test im Framework durchführt.

Auf sehr ähnliche Weise wurden auch alle anderen Kontextbedingungen geprüft. Man schreibt zuerst ein Programm, welches die Kontextbedingungen verletzt - manchmal auch auf mehrere Arten und Weisen. Danach erwartet man vom Compiler den dazugehörigen Fehler an der passenden Position im Programm.

Um jedoch die Funktionsweise von Programmen ohne Fehler systematisch zu überprüfen, wurde stark auf automatisch generierte Testfälle gesetzt, um möglichst viele vorkommende Fälle abdecken zu können, wie in den nächsten Kapiteln näher besprochen wird.

5.1 Vererbung

Als erstes wird überprüft, ob die Vererbung von Feldern und die Zuweisungskompatibilität bei Klassen korrekt funktioniert. Dazu werden 5 Klassen definiert, welche verschiedene Möglichkeiten von Vererbungen darstellen:

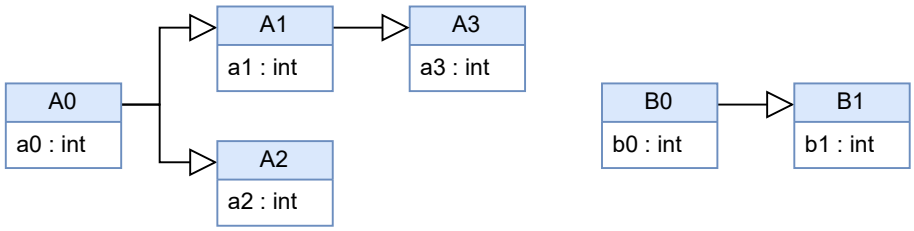


Abbildung 5.1: Klassendiagramm für Testklassen.

Danach wird ein kurzes MicroJava-Programm mit Platzhaltern (hier %s) für die spätere automatische Generierung geschrieben:

```
1 void main() %s x; { x = %s; }
```

Anschließend werden systematisch alle möglichen Kombinationen mit diesen Klassen durchgegangen, da bei Zuweisungskompatibilität im Wesentlichen nur der statische und der dynamische Typ variieren kann. Dazu wird folgende Matrix angelegt, in welcher

definiert ist, was bei einer bestimmten Kombination passieren soll. Diese Matrix wurde auch als zweidimensionales boolesches Array im Testcode gespeichert. Durch diesen Test werden insgesamt 36 Testfälle abgedeckt.

		dynamischer Typ					
		new A0	new A1	new A2	new A3	new B0	new B1
statischer Typ	A0	ok	ok	ok	ok	error	error
	A1	error	ok	error	ok	error	error
	A2	error	error	ok	error	error	error
	A3	error	error	error	ok	error	error
	B0	error	error	error	error	ok	ok
	B1	error	error	error	error	error	ok

Tabelle 5.1: Testmatrix für Vererbungstests.

Im Fall 'error' wird ein Fehler erwartet, der aussagt, dass diese Typen nicht zuweisungskompatibel sind. Im Fall 'ok' werden für alle Felder aufsteigende Zahlenwerte (von 1 weg) gesetzt und anschließend wieder ausgegeben. Für den Fall, dass sowohl statischer als auch dynamischer Typ gleich `B1` sind, wird also folgender Code generiert, wobei dementsprechend die Ausgabe '12' erwartet wird.

```
1 x = new B1;
2 x.b0 = 1;
3 x.b1 = 2;
4 print(x.b0);
5 print(x.b1);
```

5.2 Objektmethoden

Damit die Funktionsweise von Objektmethoden überprüft werden kann, müssen einige Fälle abgedeckt werden. Wiederum spielt der statische und dynamische Typ eine Rolle, weswegen die bekannten Klassen `A0`, `A1`, `A2` und `A3` aus den vorherigen Tests verwendet werden. Diesmal werden die Felder der Klassen aber nicht direkt angesprochen, sondern mithilfe von neu eingefügten Setter und Getter Methoden. Zusätzlich erhält jede dieser Klassen eine '`addAll`' Methode, mit der die Summe aller Felder gemeinsam mit einem Parameter zurückgegeben wird, für `A2` würde dies also wie folgt aussehen:

```
int addAll(int value) {
    print('2');
    return this.getA0() + this.getA2() + value;
}
```

Wie zu erkennen ist, gibt jede Klasse dabei vorher ihre Nummer aus und zählt dann die Felder der Werte anhand der Getter Methoden gemeinsam mit dem Parameter zusammen. Somit kann überprüft werden, ob die richtige Objektmethode aufgerufen wird, ob die eigenen Getter als auch die der Basisklasse gefunden werden, ob die Feldzugriffe der jeweiligen Getter die korrekten Werte zurückgeben und ob die Parameterübergabe funktioniert. In der Klasse `A2` wird dabei überall explizit `this` verwendet, in den anderen Klassen wird es jeweils nur implizit benutzt. Für die Tests wird nun wiederum eine Matrix definiert:

		dynamischer Typ			
		new A0	new A1	new A2	new A3
stat. Typ	A0	0#3	1#3	2#3	3#3
	A1	-	1#6	-	3#6
	A2	-	-	2#6	-
	A3	-	-	-	3#10

Tabelle 5.2: Testmatrix für Objektmethoden.

Die Einträge dieser Matrix stellen die erwarteten Programmausgaben dar. Die grau hinterlegten '-' Einträge können aufgrund der Zuweisungskompatibilität nicht getestet werden. Ähnlich wie im vorherigen Kapitel wird für alle anderen Fälle jeweils automatisch ein kleines Testprogramm generiert, für den Fall, dass sowohl statischer als auch dynamischer Typ gleich `A2` sind, würde dieses so aussehen:

```
x = new A2;
x.setA0(1);
if (x.a0 != x.getA0() || x.a0 != 1) { print('!'); }
x.setA2(2);
if (x.a2 != x.getA2() || x.a2 != 2) { print('!'); }
print('#');
print(x.addAll(3));
```

Es werden hier anhand des statischen Typs die Getter und Setter getestet und abschließend wird die überschriebene Methode `addAll` aufgerufen. Die erste ausgegebene Zahl soll dann laut Matrix dem dynamischen Typ entsprechen, die zweite Zahl muss die korrekte Summe abbilden.

Da die überschriebene Methode bis jetzt immer in der Wurzel des Vererbungsbaumes definiert war, wurde noch ein zweites Set an Tests geschrieben, bei der die Klasse `A1` eine `multAll` Methode definiert, welche in `A3` überschrieben wird.

		dynamischer Typ			
		new A0	new A1	new A2	new A3
stat. Typ	A0	error	error	error	error
	A1	-	1#6	-	3#0
	A2	-	-	error	-
	A3	-	-	-	3#24

Tabelle 5.3: Zweite Testmatrix für Objektmethoden.

Der Rest des Tests ist sehr ähnlich wie vorher aufgebaut, nur wird anstatt der `addAll` die neue `multAll` Methode am Schluss verwendet. Dadurch gibt es den neuen Fall 'error', da diese neue Methode in `A0` und `A2` nicht definiert ist.

5.3 Typtest und Cast

Um die Korrektheit des Typtests zu überprüfen, müssen drei Arten von Typen variiert werden: der statische Typ des zu überprüfenden Objekts, der dynamische Typ dieses Objekts und der Überprüfungstyp. Als Testklassen werden hierfür wieder dieselben wie schon in Kapitel 5.1 verwendet. Wiederum wird auch ein einfacher Beispielcode in MicroJava mit Platzhaltern geschrieben.

```
1 void main() %s x; {
2     x = %s;
3     if (x instanceof %s) { print('t'); }
4     else { print('f'); }
5 }
```

Anschließend definiert man eine Testmatrix, in welcher der Ausgang für jede Kombinationsmöglichkeit hinterlegt ist. Zu beachten ist dabei, dass die Kombinationsmöglichkeiten des statischen Typs laut Matrix so eingeschränkt wurden, dass die Zuweisungskompatibilität gewährt ist, welche ja bereits getestet wurde.

		Überprüfungstyp							
		A0	A1	A2	A3	B0	B1		
dynamischer Typ	new A0	true	false	false	false	error	error	A0	statischer Typ
	new A1	true	true	false	false	error	error	A0	
	new A1	true	true	error	false	error	error	A1	
	new A2	true	false	true	false	error	error	A0	
	new A2	true	error	true	error	error	error	A2	
	new A3	true	true	false	true	error	error	A0	
	new A3	true	true	error	true	error	error	A1, A3	
	new B0	error	error	error	error	true	false	B0	
	new B1	error	error	error	error	true	true	B0, B1	
	null	true	true	true	true	error	error	A0	
	null	error	error	error	error	true	true	B0, B1	

Tabelle 5.4: Testmatrix für Typtest und Cast.

Im Fall 'error' wird ein Fehler vom Compiler erwartet, dass diese Typen nicht verwandt sind und somit der Typtest niemals `true` liefern wird. Interessant ist dabei, dass abhängig vom statischen Typ teilweise bereits dieser Fehler erzeugt werden kann und teilweise erst zur Laufzeit das Ergebnis `false` geliefert wird. Insgesamt entstehen durch diesen Test 84 Fälle für die Überprüfung von `instanceof`.

Die Funktionalität von Cast wurde auf sehr ähnliche Weise überprüft. Der größte Unterschied ist, dass es bei Casts zu einem Laufzeitfehler kommt, welcher das Programm beendet, wo vorher der Typtest das Resultat `false` geliefert hat.

5.4 Ausnahmebehandlung

Um die Ausnahmebehandlung zu überprüfen, ist es vor allem wichtig, dass man bei einer Ausnahme im richtigen catch-Block landet, dass der Codefluss ansonsten wie gewöhnlich abläuft und dass dies auch über mehrere Methoden hinweg funktioniert. Dazu wurden die 3 Methoden `main`, `foo` und `bar` definiert, welche alle sehr ähnlich aufgebaut sind.

```
void main() %s x; %s y; {
    try {
        foo();
        print('m');
        %s // placeholder for throw statement or left empty
    } catch(x) {
        print('4');
    } catch(y) {
        print('5');
    }
}
```

Man erkennt, dass jede Methode das erste Zeichen ihres Namens und jeder catch-Block eine eigene Nummer ausgibt. Die Methode `foo` ist dabei sehr ähnlich aufgebaut und ruft ihrerseits die Methode `bar` auf, welche wiederum ähnlich ist aber nur einen catch-Block und keine weiteren Methodenaufrufe enthält, um die Anzahl der Möglichkeiten später zu reduzieren. Als Klassen zum Werfen werden dieselben aus Kapitel 5.1 verwendet, jedoch ohne `B0` und `B1`. Nun werden alle Möglichkeiten durchgegangen - jeder der 3 catch-Blöcke fängt eine der 4 möglichen Klassen und in jeder Methode wird entweder eine dieser Klassen oder nichts geworfen, wodurch sich 5 verschiedene Möglichkeiten ergeben. Bei 3 Wurforten mit 5 möglichen Ausprägungen und 5 Fangorten für eine der 4 Klassen ergeben sich also

$$5^3 * 4^5 = 128000$$

mögliche Fälle. Einige dieser Fälle sind für ein Programm nicht wirklich sinnvoll, z.B. wenn die selbe Klasse zweimal in einem try-catch-Ausdruck oder der Basistyp vor dem spezielleren Typ gefangen wird. Da der Compiler diese Varianten aber zulässt, sollten sie trotzdem auf ihre Korrektheit überprüft werden. Bei dieser Anzahl an Möglichkeiten ist es nun aber nicht mehr praktikabel, alle Fälle händisch zu überprüfen, weswegen diese Prüfung ebenfalls automatisiert stattfinden muss. Dabei kann es jedoch passieren, dass man einen möglichen Fehler des Compilers oder des Interpreters noch einmal in der Prüfungsmethode wiederholt und somit alle Tests fälschlicherweise gültig markiert werden.

Um diesen Umstand zu lösen, wurde zuerst versucht, die Kontroll-Ausgaben für die Tests mithilfe von Java zu erzeugen. In dem man das Testprogramm leicht auf den Java

Syntax umschreibt und die Testfälle mit der gleichen Methode wie für den MicroJava Compiler erzeugt, bekommt man so die gewünschten korrekten Ausgaben von Java. Jedoch funktioniert dies nicht sehr gut, da Java viele Fälle schon beim Kompilieren nicht erlaubt - zum Beispiel wenn man versucht die gleiche Klasse zweimal zu fangen oder wenn eine Klasse gefangen wird, welche nie geworfen wurde etc. Da der MicroJava Compiler diese Fälle aber nicht extra berücksichtigt, müssen sie beim Ausführen ebenso korrekt funktionieren. Deshalb wurde Python zum Generieren der Kontroll-Ausgaben verwendet, da Python diese Fälle ebenso zulässt. Das Test-Programm wurde also für Python umgeschrieben, wiederum mit Platzhaltern:

```
1 def main():
2     try:
3         foo()
4         print('m', end='')
5         %s # placeholder for throw statement or left empty
6     except %s:
7         print('4', end='')
8     except %s:
9         print('5', end='')
```

In Python wurden ebenfalls die Ausnahme-Klassen mit derselben Vererbungshierarchie definiert. Die Python Testfälle wurden dann nach demselben Prinzip wie für MicroJava erstellt und konnten dank der `exec` Funktion auch direkt von Python ausgeführt werden. Die Ausgaben von allen Python Tests werden dabei in einer Datei aggregiert. Diese Datei wird schlussendlich im MicroJava Unit-Test eingelesen, wodurch die Ausgaben von MicroJava mit denen von Python verglichen werden können. Der Inhalt der Datei sieht dabei sinngemäß wie folgt aus:

```
b1f2m4 b2m4 b2m4 b2m4 b1f3m4 b3m4 ...
```

Sollte eine Ausnahme nirgends gefangen werden, so fügt der Python Code ein Minus am Ende ein (z.B. `b1f-`), wodurch der Test für MicroJava weiß, dass nun ein Laufzeitfehler für nicht gefangene Ausnahmen zu erwarten ist.

6 Ausblick

Durch die in dieser Arbeit zu MicroJava neu hinzugefügten Funktionen ist die Sprache um ein gutes Stück mächtiger und benutzerfreundlicher geworden, aber es bleiben trotzdem einige Wünsche an eine gute Programmiersprache offen. In diesem Kapitel soll ein kurzer Ausblick gegeben werden, um welche Funktionen MicroJava in Zukunft erweitert werden könnte. Dabei ist zu beachten, dass MicroJava vor allem als Sprache zum Lernen von Compilerbau an der JKU gedacht ist. Das heißt, dass zusätzliche Funktionen möglichst neue und interessante Einblicke über den Aufbau und die Arbeitsweise von Compilern liefern sollten. Das Hinzufügen von mehr Basisdatentypen wie `short`, `float` etc. wäre hingegen eine eher repetitive Aufgabe ohne wirklich neue Erkenntnisse zu erzeugen.

Einer der am schnellsten auffallenden Unterschiede von MicroJava zu Java oder auch anderen alltäglichen Programmiersprachen ist die Syntax bezüglich der Deklaration von lokalen Variablen in Methoden. In MicroJava müssen alle lokalen Variablen bereits am Anfang der Methode deklariert werden, wodurch das Programmieren mit MicroJava etwas kontraintuitiv ist. Um dies zu ändern, müsste wiederum die Grammatik angepasst werden und eine andere Behandlung von lokalen Variablen in der Symbolliste wäre erforderlich. Außerdem würden nun auch die Blöcke von `if`- und `while`-Statements jeweils einen neuen Scope aufmachen. Die Adressvergabe an lokale Variablen wäre dadurch ebenfalls etwas schwieriger, vor allem da die Adressen von in bereits geschlossenen Scopes deklarierten Variablen wiederverwendet werden können.

Eine weitere Limitation von MicroJava ist, dass derzeit das gesamte Programm in einer einzigen Datei definiert werden muss, was für größere Projekte sehr ungünstig im Hinblick auf Organisation und Zusammenarbeit ist. Das Aufteilen von Programmen auf mehrere Dateien würde einige Änderungen im Compiler erfordern. Das MicroJava-Schlüsselwort `program` würde auf `module` geändert werden und es könnten mit einem `import` Befehl andere Module importiert werden. Als Vereinfachung zu Java enthält jede Datei also genau ein Modul und es könnten jeweils nur gesamte Module importiert werden. Jede Datei

würde danach getrennt übersetzt werden, wobei wie in Java trotzdem eine Typprüfung stattfinden sollte. Um dies zu ermöglichen, müssten die relevanten Teile der Symbolliste eines Moduls ebenfalls gespeichert werden. Dies könnte in einer eigenen Datei oder wie in Java gemeinsam mit dem Bytecode stattfinden.[8] Beim Importieren von Modulen werden diese Symbollisten anschließend geladen, wodurch die importierten Klassen und Methoden dem Compiler bekannt werden. Eine weitere Schwierigkeit wäre die erweiterte Behandlung in der μJVM, da diese nun ebenfalls mehrere Objektdateien laden und ausführen muss. Vor allem Methodenaufrufe bräuchten dadurch ein neues Konzept wie etwa dynamische Verlinkung, da die derzeitigen Sprünge zu vorher bekannten Adressen nicht länger funktionieren würden, wenn zwei oder mehr Module geladen werden.

Eine zusätzliche spannende Funktion wäre das Einfügen eines `switch`-Statements, da hier oft direkt zum Ziel anhand einer Sprungtabelle gesprungen wird. Das Benutzen so einer Tabelle ist jedoch nur sinnvoll, wenn die Werte der einzelnen Fälle nah beieinander liegen, da sonst die Tabelle zu viel Platz brauchen würde. In solch einem Fall werden die Sprünge dann ähnlich zu einem `if`-Statement erzeugt. Der tatsächlich generierte Bytecode eines `switch`-Statements ist also sehr vom jeweiligen Compiler abhängig, wodurch sich viele verschiedene Implementierungs- und Optimierungsmöglichkeiten ergeben würden.

Damit seien drei Erweiterungsideen kurz angesprochen, es gäbe natürlich auch noch viele weitere Alternativen wie überladene Methoden, Interfaces, Generics etc. Moderne Sprachen bieten zahlreiche solcher Funktionen mit denen MicroJava erweitert werden könnte. Ab einem gewissen Punkt würde dies jedoch den Zweck von MicroJava als einfaches Lehrmittel widersprechen, denn wie in dieser Arbeit geschildert, können auch Funktionen, die heute als selbstverständlich angenommen werden, die Komplexität eines Compilers deutlich erhöhen.

Literatur

[1] *Compiler Construction Handouts*. 2. Jän. 2023. URL: https://ssw.jku.at/Misc/CC/Handouts.pdf (besucht am 25. 08. 2023) (siehe S. 3).

[2] *The Compiler Generator Coco/R*. 3. Dez. 2018. URL: https://ssw.jku.at/Research/Projects/Coco/ (besucht am 07. 08. 2023) (siehe S. 4).

[3] *Java Reference Type Casting*. 20. Juni 2022. URL: https://docs.oracle.com/javase/specs/jls/se8/html/jls-5.html#jls-5.5.1 (besucht am 27. 08. 2023) (siehe S. 6, 21).

[4] *Java NullPointerException*. 20. März 2023. URL: https://docs.oracle.com/en/java/javase/20/docs/api/java.base/java/lang/NullPointerException.html (besucht am 27. 08. 2023) (siehe S. 20).

[5] *Java ClassCastException*. 20. März 2023. URL: https://docs.oracle.com/en/java/javase/20/docs/api/java.base/java/lang/ClassCastException.html (besucht am 27. 08. 2023) (siehe S. 22).

[6] *The Java Virtual Machine Instruction Set - instanceof*. 20. März 2023. URL: https://docs.oracle.com/javase/specs/jvms/se20/html/jvms-6.html#jvms-6.5.instanceof (besucht am 27. 08. 2023) (siehe S. 23).

[7] *The Java Language Specification*. 31. Aug. 2022. URL: https://docs.oracle.com/javase/specs/jls/se19/jls19.pdf (besucht am 30. 08. 2023) (siehe S. 25).

[8] *The class File Format*. 20. März 2023. URL: https://docs.oracle.com/javase/specs/jvms/se20/html/jvms-4.html (besucht am 27. 08. 2023) (siehe S. 40).