Heinrich Heine
Universität
Düsseldorf

INSTITUT FÜR INFORMATIK
Softwaretechnik und Programmiersprachen
Universitätsstr. 1 D–40225 Düsseldorf

Visualisieren von Algorithmen in Compilern

Tom Schreiner

Bachelorarbeit

Beginn der Arbeit:	05. November 2024
Abgabe der Arbeit:	05. Februar 2025
Gutachter:	John Witulski
	Fabian Ruhland

Ehrenwörtliche Erklärung

Hiermit versichere ich die vorliegende Bachelorarbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt zu haben. Alle Stellen, die aus den Quellen entnommen wurden, sind als solche kenntlich gemacht worden. Diese Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

Düsseldorf, den 05. Februar 2025

Tom Schreiner

Zusammenfassung

Das Ziel dieser Bachelorarbeit war es, einige Algorithmen und Konzepte aus dem Compilerbau zu visualisieren um Personen, insbesondere Studenten, eine Hilfe beim Studieren dieser zu geben.

Dafür wurde ein Framework entwickelt, welches diese und beliebige weitere Algorithmen verständlich und Schritt für Schritt visualisieren kann.

Folgende Algorithmen wurden implementiert:

1. Analyse von erreichenden Definitionen für Grundblöcke
2. Liveness Analyse für Grundblöcke
3. Liveness Analyse für einzelne 3-Address-Code Instruktionen
4. Erstellen von Grundblöcken für ein Programm geschrieben in 3-Address-Code
5. Erstellen eines Kontrollflussgraphen für ein 3-Address-Code Programm

Um diese Algorithmen simpel und gut verständlich zu visualisieren, brauchte das Framework zwei Arten von Darstellungen:

1. Graphen: um Kontrollflussgraphen darzustellen
2. Tabellen: um Datenflusswerte darzustellen und 3-Address-Code in einer angenehmen Art und Weise zu visualisieren

Desweiteren brauchte es die Möglichkeit, 3-Address-Code und Grundblöcke einfach zu verarbeiten.

Außerdem ist es durch Implementierung eines Interfaces einfach, weitere Algorithmen hinzuzufügen. Es wurden weitere Interfaces implementiert, mit denen häufig genutzte Funktionalitäten, wie zum Beispiel ein Button, um Code aus einer Datei zu laden, einfach in neuen Plugins genutzt werden können. Somit können weitere Plugins mit sehr viel weniger Aufwand hinzugefügt werden.

Inhaltsverzeichnis

1 Einleitung

1.1 Motivation

Als ich im Wintersemester das Modul "Compilerbau" belegt habe, habe ich mich das erste Mal mit einigen Algorithmen beschäftigt, die eben in diesem Themengebiet angewendet werden. Dabei fiel es mir bei einigen schwer, mir diese ohne weiteres vorzustellen. Ein Tool, mit dem man Schritt für Schritt durch diese Algorithmen gehen kann, hätte mir das Verstehen der Algorithmen und vor allem das Entwickeln einer Intuition warum diese Algorithmen überhaupt so funktionieren wie sie es tun stark erleichtert.

Es gibt zwar solche Tools, zum Beispiel VisOpt[fra] oder DFAV[Kasa], diese benutzen allerdings ein Subset von Java und JavaScript. Da im Modul "Compilerbaußämtliche Algorithmen auf drei Address Code erklärt werden, passen diese Tools nicht in den Usecase der Studenten.

In dieser Bachelorarbeit wird ein Framework in Java entwickelt, mit dem es einfach sein soll, Algorithmen zu visualisieren.

Im Jahr 2016 wurde von Fabian Ruhland und Isabel Wingen bereits ein ähnliches Framework entwickelt[FR]. Dieses ist leider mittlerweile nicht mehr mit aktuellen Java-Versionen kompatibel, daher wurde sich dazu entschieden ein komplett neues Framework zu entwickeln.

Zudem werden einige der Algorithmen, die im Compilerbau verwendet werden, implementiert. Das Framework basiert darauf, dass durch Implementierung von Interfaces einfach neue Algorithmen als Plugins hinzugefügt werden können. Dadurch kann gewährleistet werden, dass wenn etwas im Framework nicht mehr funktioniert, zum Beispiel durch neue Versionen von Java oder einem Update einer Dependency, dieses Modul ausgetauscht werden kann ohne die implementierten Plugins aktualisieren zu müssen.

1.2 Theoretische Grundlagen

Im folgenden Abschnitt werden die für diese Arbeit notwendigen grundlegenden Konzepte und Algorithmen erklärt, da im weiteren Verlauf der Arbeit nur auf die Implementierung dieser eingegangen wird.

1.2.1 Drei Address Code

Drei Address Code ist eine Art von Zwischencode [1]. Charakterisierend für Drei Address Code (folgend auch 3-Address-Code genannt) ist, dass einzelne Instruktionen auf maximal drei Addressen(oder Konstanten) zugreifen. Die Addresse in die der resultierende Wert gespeichert wird und eine oder zwei Addressen oder Konstanten aus denen sich der resultierende Wert bildet. Dazu ist noch die Operation, die ausgeführt wird angegeben.

Für diese Bachelorarbeit wurde eine Teilmenge des im Drachenbuch [Aho08, Kapitel 6.2.1] beschriebenen 3-Address-Code verwendet.
Folgende Operationen gibt es:

1. Binäre Operationen X = Y op Z
   in denen das Resultat aus einer der folgenden binären Operation in der Addresse X gespeichert wird. Implementiert wurden Addition, Subtraktion, Multiplikation und Division.

2. Die unäre Operation X = - Y
   in der der invertierte Wert von Y in X gespeichert wird.

3. Der Kopierbefehl X = Y
   in der der Wert der in Addresse Y gespeichert ist in X kopiert wird.

4. Der unbedingte Sprung goto X
   hier wird kein Wert gespeichert, sondern zu der Addresse die in X gespeichert ist gesprungen

5. Die bedingten Sprünge if Y goto X und ifFalse Y goto X
   mit denen, wenn der Wert Y entweder $wahr$ oder $falsch$ repräsentiert zur Addresse X gesprungen wird oder die nächste Instruktion ausgeführt wird, wenn dies nicht der Fall ist.

[1] Zwischencode hat viele Nutzungsgebiete in einem Compiler, einerseits verbindet es das Frontend (welche den Quellcode einließt) mit dem Backend (welche den Maschinencode ausgibt), andererseits ist Zwischencode plattformunabhängig, sodass maschinenunabhängige Optimierungen und Analysen auf diesem ausgeführt werden können.

6. Die bedingten Sprünge if Y relOp Z goto X
   in denen auch zu X gesprungen wird, wenn die Relation Y relOp Z $wahr$ ist, sonst wird auch hier die nächste Instruktion ausgeführt.
   Die Implementierten Relationen sind:
   - Y $<$ Z
   - Y $\leq$ Z
   - Y $>$ Z
   - Y $\geq$
   - Y $=$ Z
   - Y $\neq$ Z

Hierbei können die Addressen X, Y und Z beliebige Zeichenfolgen sein, Y und Z können außerdem Konstanten sein. Die einzelnen Elemente jeder Instruktion sind durch ein Leerzeichen von einander getrennt. Verschiedene Instruktionenen werden durch einen Zeilenumbruch getrennt. Da sich die Algorithmen in dieser Arbeit nicht mit komplexeren Aufgaben wie Speichermanagement beschäftigen, wurde sich dagegen entschieden den 3AC umfangreicher zu modellieren.

Ein Beispiel für gültigen 3-Address-Code wäre also:

Listing 1: 3-Address-Code, der die 5-te Fibonacci Zahl ausrechnet und in x speichert

```
 0: n = 5
 1: fib = 1
 2: lst = 1
 3: n = n - 2
 4: if n <= 0 goto 10
 5: hlp = lst
 6: lst = fib
 7: fib = lst + hlp
 8: n = n - 1
 9: goto 4
10: x = fib
```

1.2.2 Grundblöcke

Grundblöcke sind Instruktionsfolgen eines Zwischencodeprogrammes, die immer zusammen ausgeführt werden. [Aho08, S.619] Dies ermöglicht es, einen Block an Instruktionen anzuschauen und bestimmte Optimierungsalgorithmen auf sie anzuwenden, ohne Gefahr zu laufen, die Semantik des Programmes zu verändern.

Um ein Programm in Grundblöcke aufzuteilen, kann man wie folgt vorgehen[Aho08, S.643]:

1. Markiere die erste Instruktion des Programmes als Leader, da diese immer ausgeführt wird.
2. Markiere alle Instruktionen als Leader, die Ziel eines Sprungs sind oder auf einen Sprung folgen.
3. Alle Instruktionen die auf eine markierte Instruktion folgen, bis zu einer neuen markierten Instruktion gelten nun als ein Grundblock.

Folgend kann man für jeden Grundblock bestimmen, welche Grundblöcke auf ihn folgen, daraus lässt sich ein Flussgraph, bestimmen den wir, da er den Kontrollfluss beschreibt, folglich Kontrollflussgraphen nennen werden. In diesem stellt jeder Grundblock einen Knoten dar und jede Kante einen Sprung von einem Grundblock zum anderen.

Aus unserem Beispielcode aus dem Letzten Kapitel können wir also folgende Instruktionen markieren:

Listing 2: Fibonacci 3-Address-Code mit markierten Leadern

```
 0: n = 5 //Leader, da erste Instruktion
 1: fib = 1
 2: lst = 1
 3: n = n - 2
 4: if n <= 0 goto 10 //Leader, da 9 hierhin springt
 5: hlp = lst //Leader, da 4 ein bedingter Sprung ist
 6: lst = fib
 7: fib = lst + hlp
 8: n = n - 1
 9: goto 4
10: x = fib //Leader, da 9 ein Sprung ist und 4 hierhin springen kann
```

Daraus folgt, dass wir folgende Grundblöcke haben:

1. Ein Grundblock B_0 der die Addressen 0 bis 3 besitzt.
2. Der Block B_1 der nur die Addresse 4 hat.
3. Grundblock B_2 der die Addressen 5 bis 9 besitzt.
4. und Block B_3 der nur die Addresse 10 besitzt.

1.2.3 Kontrollflussgraphen

Kontrollflussgraphen sind gerichtete Graphen, deren Knoten aus Grundblöcken und deren Kanten aus den jeweilig folgenden Grundblöcken besteht.

Um unser Beispiel weiterzuführen, bildet sich folgender Kontrollflussgraph:

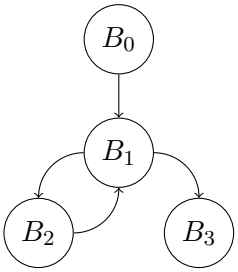

Abbildung 1: Resultierender Kontrollflussgraph

Auf Kontrollflussgraphen lassen sich nun sogenannte Datenflusswerte definieren. Diese sind Abstraktionen für die Menge aller möglichen Zustände des Programmes zu einem bestimmten Zeitpunkt. In Bezug zu einem Kontrollflussgraphen ist dieser Zeitpunkt immer entweder (direkt) bevor der Ausführung des Grundblockes, die $in[B]$ Menge, oder nach der Ausführung dessen, die $out[b]$ Menge. Je nach dem was wir analysieren beziehungsweise abstrahieren wollen ändern sich die Datenflusswerte, da nur die für die jeweilige Analyse relevanten Informationen betrachtet wird. Ausserdem abhängig von der spezifischen Analyse ist die Flussrichtung der Datenflusswerte. Wenn der Datenfluss vorwärts gerichtet ist, verändert sich die $out[B]$ Menge durch Anwendung einer sogenannten Transferfunktion auf der $in[B]$ Menge, also gilt:

$$out[B] = f_B(in[B])$$

$$in[B] = \bigcup_{v \in V_B} out[v]$$

Analog gilt, wenn der Datenfluss rückwärtsgerichtet ist:

$$in[B] = f_b(out[B])$$

$$out[B] = \bigcup_{n \in N_B} in[n]$$

Hierbei sind V_B und N_B die Mengen der Vorgänger und Nachfolger eines Grundblockes B.[Aho08, S.732-734]

1.2.4 Erreichende Definitionen

Dies ist eine der gebräuchlichsten und nützlichsten Datenflussanalysen[Aho08, S.734]. Mit ihnen können wir herausfinden, welchen Variablen zu einem bestimmten Zeitpunkt ein Wert zugewiesen ist. Eine Anwendung wäre zum Beispiel zu kontrollieren, ob eine Variable zu einem bestimmten Zeitpunkt überhaupt einen Wert hat[Aho08, S.734], sofern die ursprüngliche Programmiersprache dies als notwendig erachtet. Man kann aber auch schauen, ob die Variable eine Konstante ist und somit Instruktionen gespart werden können.

Für die Berechnung der erreichenden Definitionen bestimmen wir folgende Datenflusswerte für jeden Grundblock:

- Die gen_B Menge:
  Beschreibt die Addressen aller Instruktionen in einem Grundblock B welche einer Variable einen Wert zuweisen, also einen Wert *generieren*. Wenn einer Variable ein Wert mehrmals zugewiesen wurde liegt nur die letzte Zuweisung dieser in der gen_B Menge.
- Und die $kill_B$ Menge:
  Beschreibt die Addressen aller Instruktionen welche einer Addresse einen Wert zuweisen, der durch eine Instruktion in der gen_B Menge überschrieben wird.

Nun können wir uns herleiten dass alle Variablen die in einem Grundblock B definiert werden, ihn auch verlassen also Teil der $out[B]$ Menge sind. Zudem verlassen auch alle Definitionen den Grundblock, wenn sie nicht überschrieben wurden. Demnach ist der Datenfluss vorwärtsgerichtet und es gilt die Transferfunktion:

$$out[B] = gen_B \cup (in[B] \backslash kill_B)$$

Hierbei handelt es sich um einen Fixpunktalgorithmus. Das heißt, wir beginnen mit $out[B] = \emptyset$ für alle Grundblöcke und iterieren dann so lange über diese, bis sich keine Menge mehr verändert.[Aho08, S.739-740]

Für unser Beispiel ergibt sich dann Tabelle 1:

B	B_0	B_1	B_2	B_3
gen_B	$1, 2, 3$		$5, 6, 7, 8$	10
$kill_B$	$0, 6, 7, 8$		$1, 2$	
$in_1[B]$				
$out_1[B]$	n, fib, lst		n, fib, lst, hlp	x
$in_2[B]$		n, fib, lst, hlp		
$out_2[B]$	n, fib, lst	n, fib, lst, hlp	n, fib, lst, hlp	x
$in_3[B]$		n, fib, lst, hlp	n, fib, lst, hlp	n, fib, lst, hlp
$out_3[B]$	n, fib, lst	n, fib, lst, hlp	n, fib, lst, hlp	x, n, fib, lst, hlp
$in_4[B]$		n, fib, lst, hlp	n, fib, lst, hlp	n, fib, lst, hlp
$out_4[B]$	n, fib, lst	n, fib, lst, hlp	n, fib, lst, hlp	x, n, fib, lst, hlp

Tabelle 1: Erreichende Definitionen für Fibonacci. Veränderungen sind blau markiert

1.2.5 Lebendige Variablen

Bei der Analyse lebendiger Variablen (folgend auch liveness Analyse genannt) bringen wir in Erfahrung, ob ein bestimmter Wert zu einem bestimmten Zeitpunkt lebendig ist. Lebendig bedeutet in diesem Kontext, das dieser Wert definiert wurde und zu einem späteren Zeitpunkt im Programm auch noch genutzt wird.

Die Analyse lebendiger Variablen hat viele Anwendungsgebiete. Beispielsweise bei der Registervergabe:
Ein reeller Computer hat nur eine begrenzte Anzahl an Registern. Somit können nicht unendlich viele Variablen gleichzeitig ein Registern zur Verfügung stehen.
Im worst case Szenario bedeutet das, dass alle Werte nach ihrer Berechnung in den Speicher geschrieben werden müssen und vor jeder Berechnung aus dem Speicher geladen werden müssen.
Da dies viel mehr Zeit kostet als Werte welche wieder genutzt werden, bis dahin in einem Register zu lassen [2], ist es sinnvoll Interaktionen mit dem Speicher so gering wie möglich zu halten.
Die liveness Analyse kann hier berechnen wie viele Register wir maximal benötigen, da eventuell nicht alle Variablen gleichzeitig lebendig sind, also gleichzeitig benötigt werden.
(Frei nach dem Drachenbuch[Aho08, S.743-744] zitiert.)

[2] Wir sparen also das Speichern in einer Address und das Laden aus einer Addresse

Auch für die liveness Analyse auf Grundblöcken [3] definieren wir wieder zwei Mengen, welche wir für die Transferfunktion benötigen[Aho08, S.743]:

1. Die def_B Menge:
   In der def_B Menge sind alle Variablen enthalten, denen im Gundblock B ein Wert zugewiesen wird, bevor diese "verwendet"wird. Das bedeutet dass wir alle Variablen in dieser Menge zu Beginn des Blockes als "tot"betrachten können.
2. die use_B Menge:
   Die use_B Menge definiert alle Variablen, deren Werte vor ihrer Definition verwendet werden. Dementsprechen sind alle Variablen in dieser Menge zu beginn des Grundblockes "lebendig".

Daraus bilden wir uns eine rückwärtsgerichtete Transferfunktion:

$$in[B] = use_B \cup (out[B] \backslash def_B)$$

Da nur die Variablen zu Beginn der Ausführung von Grundblock B leben müssen welche benutzt werden, oder in einem Nachfolger benutzt werden, aber nicht in diesem Block definiert werden.

In unserem Beispiel resultiert dies in folgenden Mengen:

B	B_0	B_1	B_2	B_3
def_B	n, fib, lst		hlp	x
use_B		n	lst, fib, n	fib
$out_1[B]$				
$in_1[B]$		n	lst, fib, n	fib
$out_2[B]$	n	lst, fib, n	n	
$in_2[B]$		n, lst, fib	lst, fib, n	fib
$out_3[B]$	n, lst, fib	lst, fib, n	lst, fib, n	
$in_3[B]$		n, lst, fib	lst, fib, n	fib
$out_4[B]$	n, lst, fib	lst, fib, n	lst, fib, n	
$in_4[B]$		n, lst, fib	lst, fib, n	fib

Tabelle 2: Liveness Analyse für unser Fibonacci Programm. Veränderungen sind blau markiert

Dies sind alle Theoretischen Grundlagen die für das Verstehen der in dieser Arbeit implementierten Konzepte, wenden wir uns nun der Implementierung zu.

[3] Die liveness Analyse kann auch auf einzelnen Instruktionen ausgeführt werden. In Kapitel 3.5 wird dies erneut aufgegriffen und erläutert.

2 Framework

In diesem Abschnitt soll es um das Implementierte Framework gehen. Aus den theoretischen Grundlagen lässt sich schließen, dass folgende Daten (-Strukturen) unbedingt im Framework implementiert sein sollten:

1. Drei Address Code Instruktionen:
   Um einzelne Instruktionen zu modellieren zu können brauchen diese einen eigenen Datentyp, um Eigenschaften wie Sprünge, konstante Werte oder gelesene und geschriebene Variablen darstellen zu können.
2. Drei Address Code Operationen:
   Da es 16 verschiedene Operationen gibt, von denen sich auch noch einige gruppieren lassen, ist es sinnvoll, ein Enum zu schreiben, um mit switch-Statements arbeiten zu können.
3. Grundblöcke:
   Die Grundblockklasse soll speichern, wo im 3-Address-Code Programm einzelne Grundblöcke anfangen, aufhören und zu welchen Addressen sie springen.
4. Drei Address Code:
   Diese Klasse soll das gesamte Programm modellieren, also alle Instruktionen speichern und die Grundblöcke generieren.

Daraus ergibt sich folgendes Klassendiagramm(Abb. 2):

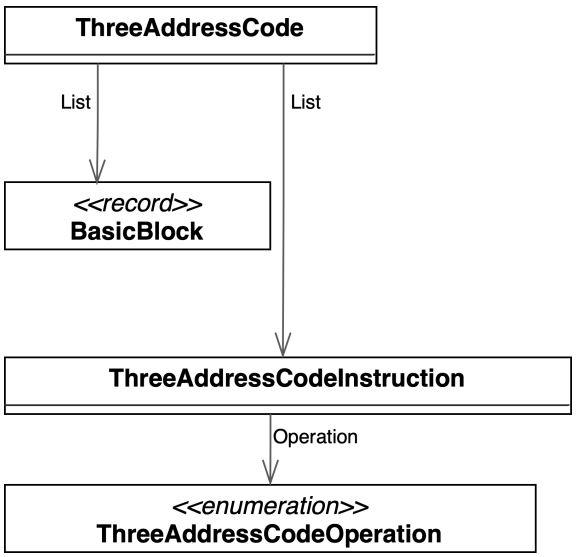

Abbildung 2: Die Drei Address Code Klassen im Überblick

Da der Usecase dieses Frameworkes das Visualisieren von Algorithmen ist, braucht es folglich eine Möglichkeit, den 3-Address-Code zu visualisieren. Hier wurde sich für eine Tabelle entschieden, da eine Instruktion in acht Zellen [4] aufgeteilt werden kann. Da es sich bei einem Programm immer um eine Liste von Instruktionen handelt, ergibt sich so ein Zweidimensionales Feld an Daten. Außerdem sollen auch die Daten die wir aus unseren Analysen erheben angezeigt werden. Auch hier ist eine Tabelle sinnvoll.

Um Kontrollflussgraphen anzeigen zu können, soll es ausserdem möglich sein Graphen anzuzeigen.

Mit einer Toolbar soll es möglich sein mit dem aktuell geladenen Plugin zu interagieren.

Aus diesen Anforderungen ergibt sich folgendes Klassendiagramm(Abb. 3):

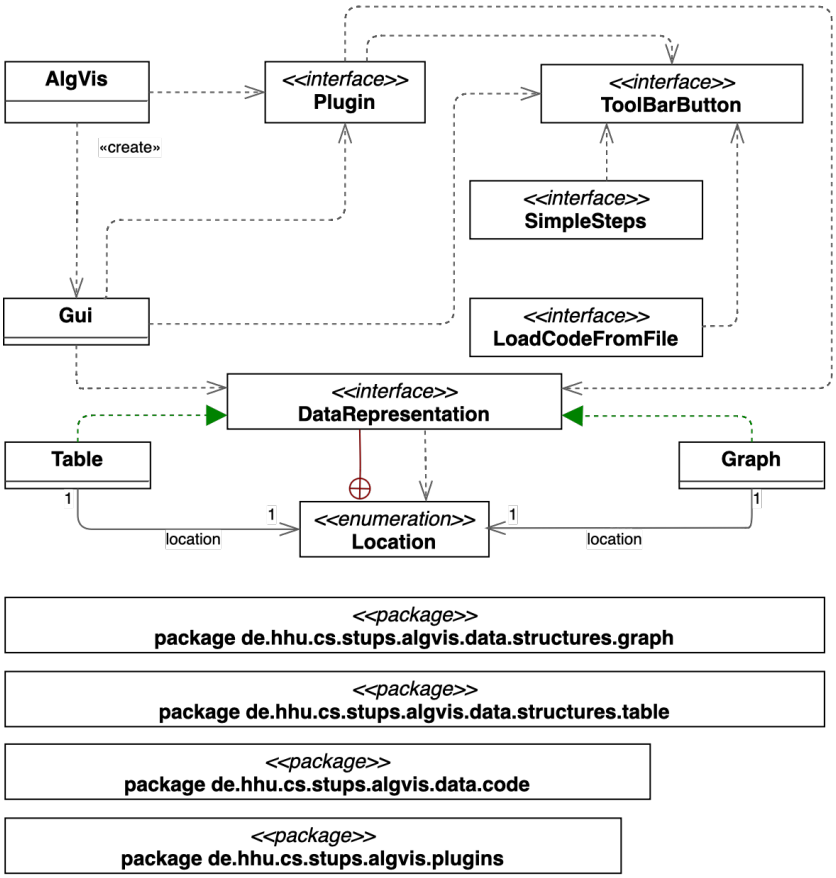

Abbildung 3: Klassenstruktur der GUI-Klassen

[4] $if\ Y\ relOp\ X\ goto\ L$ ist mit 6 Elementen die längste legale 3-Address-Code Instruktion, dazu wird noch eine Zelle für die aktuelle Instruktion und eine für Kommentare hinzugefügt

- Die Klasse *AlgVis* gilt hier (Abb. 3) als Entrypoint. Sie lädt alle Plugins, erstellt ein *Gui* Objekt und übergibt diesem die Plugins.
- Das Interface *Plugin* definiert welche Methoden neue Plugins beziehungsweise Algorithmen benötigen um hinzugefügt zu werden, diese werden in *de.hhu.cs.stups.algvis.plugins* implementiert.
- Zudem definiert das Interface *ToolBarButton* wie Buttons der ToolBar, welche im nächsten Kapitel im Detail erklärt wird, implementiert werden können. Beispiele dafür liefern das *SimpleSteps* und *loadCodeFromFile* Plugin, welche vordefinierte Buttons anbieten.
- Die Packages endend auf *graph* und *table* enthalten hierbei Helferklassen für die jeweiligen Komponenten, hierzu später mehr.
- Im Package *de.hhu.cs.stups.algvis.data.code* sind die vorhin genannten Datenstrukturen für 3-Address-Code, Grundblöcke, 3-Address-Code-Instruktionen und -Operationen implementiert.

2.1 Gui

Die *Gui* Klasse ist das Herzstück der Visualisierung. Hier wird ein *JFrame*, also ein GUI Fenster der Java Standardlibrary *Swing* geladen. In ihr befinden sich drei GUI-Elemente, auch aus der *Swing*-Library:

1. Eine Menüleiste, in der über ein Dropdown alle Plugins aufgerufen werden können.
2. Ein *JPanel*, welches im folgenden *ContentPanel* genannt wird, da sich in ihm alle grafischen Elemente des aktuell genutzten Plugins befinden. In ihm können Plugins Objekte die das Interface *DataRepresentation* implementieren einfügen. Die Klassen *Table* und *Graph* implementieren dieses Interface bereits. Sollte später ein Plugin eine andere Art der Darstellung benötigen, kann dieses das Interface mit einer anderen *awt*-Komponente implementieren. Zum Start des Frameworks zeigt es einen SplashScreen (Abb. 4) mit dem Schriftzug „welcome“an.
3. Und eine *JToolBar*.
   Da die meißten Plugins ähnliche Funktionalitäten haben, wie zum Beispiel das schrittweise Durchlaufen eines Algorithmus, wurde ein GUI-Element hinzugefügt in dem Buttons zur Kontrolle des Plugins hinzugefügt werden können. Diese ist auf der Abbildung(Abb. 4) nicht zu sehen, da aktuell kein Plugin geladen ist und somit auch kein Buttons angezeigt werden

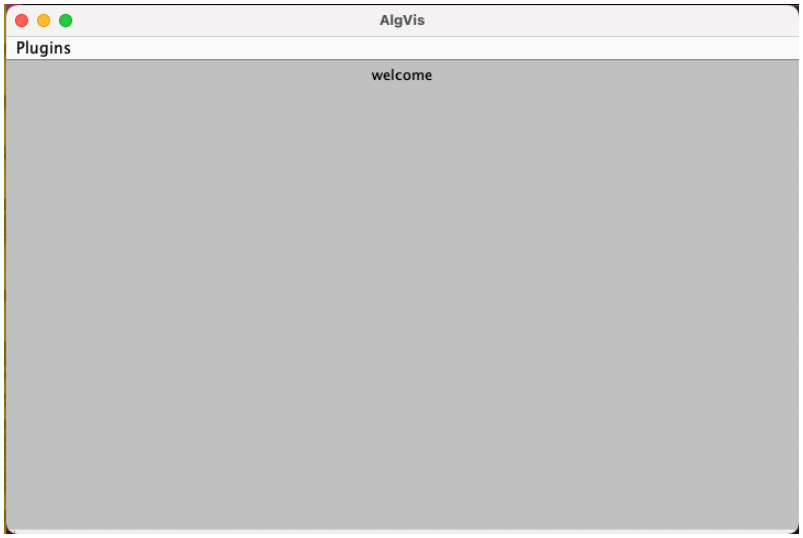

Abbildung 4: Ansicht des Programmes direkt nach Start

2.2 Darstellen von Daten

Um grafische Elemente im *ContentPanel* darzustellen wird das Interface *DataRepresentation* von Gui-Klassen implementert. Dadurch kann auf zwei Methoden zugegriffen werden:

- *getSwingComponent()* gibt die *awt*-Komponente zurück, welche dem *ContentPanel* hinzugefügt wird.
- Durch die Methode *getComponentLocation()* wird bestimmt an welcher Position im *ContentPanel* diese angezeigt wird. Dies wird

Das Enum *Location* entscheidet hierbei wo im *ContentPanel* das Gui-Objekt angezeigt wird.

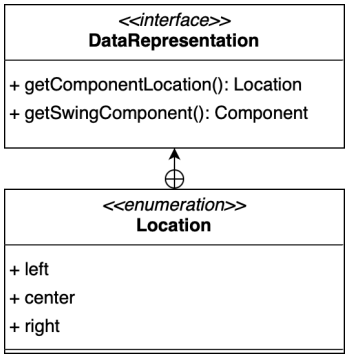

Abbildung 5: Das DataRepresentation Interface und sein Location enum

Für diese Arbeit wurden folgende Gui-Klassen implementiert:

2.2.1 Graphen

Graphen werden durch die Klasse *Graph* und zwei Subklassen *Edge* und *Node* realisiert(siehe Abb. 7). Da das Schreiben einer eigenen Engine für die Darstellung von Graphen diese Bachelorarbeit übertreffen würde wurde entschieden eine externe Library zu verwenden. Hier wurde sich für GraphStream[GTc] entschieden.

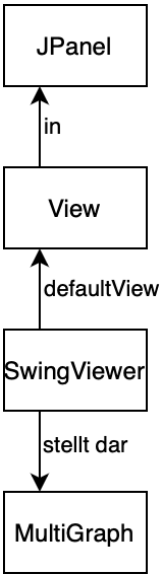

Abbildung 6: Rendering Pipeline eines Graphen

Graphstream ist eine Library zum modellieren, analysieren und visualisieren von Graphen, im Framework wird sie jedoch nur benutzt um Graphen zu visualisieren.
Um einen Graphen im *ContentPanel* anzeigen zu lassen gibt die Methode *getSwingComponent()* ein JPanel wieder, in diesem JPanel befindet sich ein GraphStream *View* Objekt welches von einem *SwingViewer* erstellt wird, welcher in einem eigenen Thread(siehe Listing 18) einen *MultiGraph* rendert(siehe Abb. 6). Dies stellt die default Renderingstrategie dar[GTb].
Graphstream bietet einige verschiedene Implementierungen von Graphen an, die je nach Usecase unterschiedliche Vorteile haben. Hier wurde sich dafür entschieden nur die *MultiGraph* implementierung zu verwenden, da es möglich sein muss mehrere Kanten zwischen denselben zwei Knoten zu haben. [5] Diese Implementierung wird laut Dokumentation nur von einem *MultiGraph* implementiert.[GTa]

Nun wird im GUI ein leerer Graph angezeigt.
Um dem *MultiGraph* nun einen Knoten hinzuzufügen benutzen wir die Methode *addNode(String)*. Diese erwartet einen String als ID, dementsprechen müssen alle Knoten die wir hinzufügen einen eindeutigen String haben.
Um eine Kante hinzuzufügen wird die Methode *addEdge(String, String, String, boolean)* hierbei soll der erste String die eindeutige ID der Kante sein und die beiden folgenden Strings die IDs der jeweiligen Aus- und Eingangsknoten. Der Boolean-Wert gibt hierbei an ob die Kante eine gerichtete Kante ist, oder nicht. Um Knoten oder Kanten zu entfernen gibt es die Methoden *removeNode(String)* und *removeEdge(String)* die nach demselben Prinzip agieren.

Um das Aussehen eines Graphen anzupassen werden Attribute benutzt[GTd]. Attribute können Graphen, Knoten und Kanten mit der Methode *setAttribute(String, Object)* hinzugefügt werden, hierbei ist der erste String das Attribut und das folgende Objekt der Wert des Attributes. Im Falle dieser Arbeit sind zwei Attribute relevant:

[5] Es wird die Möglichkeit benötigt eine Kante a>b und eine Kante b>a gleichzeitig darzustellen

- Das gerenerelle „look and feel“des Graphen wird durch das „ui.stylesheet“Attribut bestimmt. Dieses setzen wir für das Graph-Objekt mit *graph.setAttribute(„ui.stylesheet“, „[...]“)*
- Um die einzelnen Knoten von einander unterscheiden zu können, gibt es die Möglichkeit diese mit einem Text zu versehen. Das Attribut hierfür ist „ui.label“. Um einem Knoten ein Label zu geben ist unser Methodenaufruf folglich *node.setAttribute(„ui.label“, „[...]“)*. Da wir diese Methode auf einem node-Objekt aufrufen, dies aber nicht selber erstellen, benötigen wir ausserdem die Methode *graph.getNode(String)*, bei der der übergebene String wieder die ID des Knotens ist.

Um unabhängig von *GraphStream* mit Graphen umzugehen zu können wurde mit folgender(Abb. 7) Abstraktion gearbeitet:

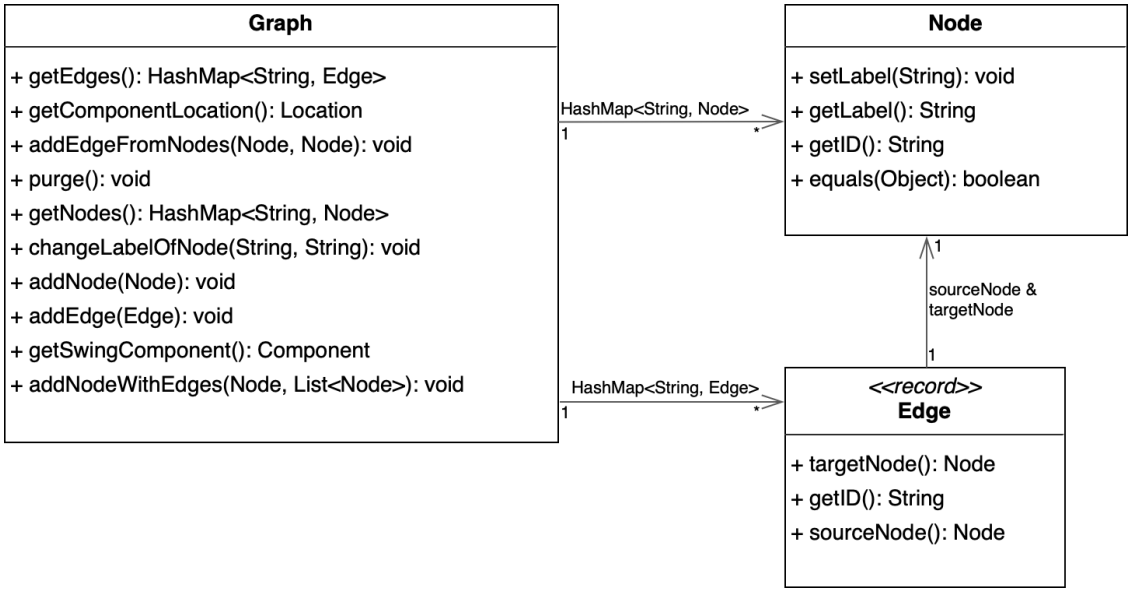

Abbildung 7: Klassenstruktur der Graph-Klassen

Hierbei sind die Klassen *Node* und *Edge* Helferklassen, um anstelle von Strings mit Node- beziehungsweise Edge-Objekten zu arbeiten.
Einem *Node*-Objekt wird zu ihrer Erstellung eine eindeutige ID zugewiesen um das Objekt später mit in einem *GraphStream*-Graphen verwenden zu können.
Ein *Edge*-Objekt ist ein Record welcher zwei Node-Objekte sourceNode und targetNode beinhält. Die für *GraphStream* eindeutige ID bestimmt sich dann durch die *getID()*-Methode(Listing 17):

Die *Graph* Klasse ist die zentrale Klasse für die Visualisierung von Graphen. Sie implementiert das *DataRepresentation* Interface und alle nötige Kommunikation mit der *GraphStream* Library.

Die im Graph enthaltenen Knoten und Kanten werden in einem *Set* gespeichert(Listing 18), wenn ein Plugin dem Graphen einen Knoten hinzufügen möchte, ruft er die Methode *addNode(Node)*(Listing 19) auf. Diese fügt den Knoten dem Set der im Graphen enthaltenen Knoten hinzu und fügt den Graphen einen neuen Knoten mit der ID des übergebenen Knotens hinzu. Anschließend wird die Methode *layout.shake()* aufgerufen. diese Methode sorgt dafür dass die Anordnung der Knoten neu berechnet wird, sodass diese immer einen angemessenen Abstand zu einander haben und sich nicht überlappen. Analog gibt es auch die Methode *addEdge(Edge)* um Kanten hinzuzufügen und die Methoden *removeEdge(Edge)* und *removeNode(Node)* um diese wieder zu entfernen.
Um den Graphen komplett zu leeren wurde die Methode *purge()* hinzugefügt. diese entfernt alle Knoten und Kanten vom Graphen.
Um einem Knoten ein Label zu geben, wurde die Methode
setLabelOfNode(Node, String)(Listing 20)hinzugefügt.

2.2.2 Tabellen

Tabellen werden durch die Klasse *Table*(Abb. 8) visualisiert. Da bereits mit der *JTable* von der *swing*-Library eine Tabellenvisualisierung angeboten wird, wurde diese auch implementiert.
Ein *JTable*-Objekt benötigt immer ein *TableModel*, welches die Daten, welche dargestellt werden sollen, enthält.
Dieses wird von der Klasse *DataTableModel*(Abb. 8) implementiert. Sie speichert die Daten in einem zweidimensionalen String-Array und implementiert alle vom *TableModel*-Interface vorgegebenen Methoden.

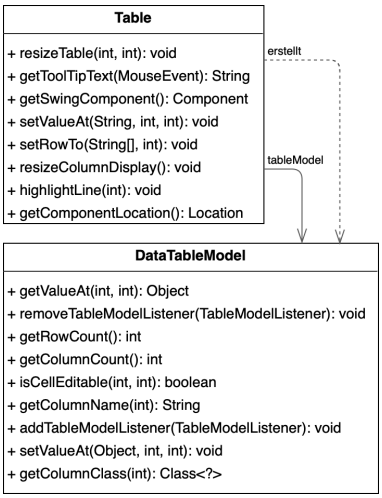

Abbildung 8: Klassenstruktur der Tabellenvisualisierung

Die notwendigen, nicht trivialen, Methoden wurden wie folgt implementiert:

- *isCellEditable(int, int)* gibt immer *false* zurück, da es nicht erwünscht ist, dass der Benutzer die Tabelle bearbeiten kann. Ein Plugin kann die Tabelle trotzdem bearbeiten.

- Die Methode *getColumnName(int)* gibt immer *null* zurück, da in aktuellen Stand des Frameworks Überschriften selber gesetzt werden.

- Die Methoden *addTableModeListener(TableModelListener)* und *removeTableModeListener(TableModelListener)* fügen eine Implementation der Klasse TableModel-

Listener einer zugehörigen Liste hinzu, beziehungsweise entfernen sie. [6]

- *setValueAt(Object, int, int)* versucht den Wert am übergebenen Index zu überschreiben. Wenn dies möglich ist werden alle Objekte in der TableModelListener Liste benachrichtigt (Listing 21).

Ein Objekt der *Table*-Klasse (Abb. 8) erweitert die von *swing* gegebene *JFrame*-Klasse und gibt sich selber in *getSwingComponent()* zurück.

Die für Plugins verfügbar gestellten Methoden sind:

- Die Methode *resizeTable(int, int)* (Listing 22) erstellt ein neues *DataTableModel*-Objekt mit der spezifizierten Größe und setzt dieses als neues *TableModel*.
- *setValueAt(String, int, int)* setzt den übergebenen String als Wert an der übergebenen Position indem es *setValueAt(Object, int, int)* im *DataTableModel* aufruft.
- *setRowTo(String[], int)* Setzt die Werte für eine gesamte Zeile indem es über das String-Array iteriert.
- *highlightLine(int)* (Listing 23) Markiert die übergebene Zeile.

Desweiteren wurden zwei weitere Funktionen implementiert, um Zelleninhalte besser zu visualisieren:

- Die Methode *getToolTipText(MouseEvent)* (siehe Listing 24) überschreibt die in *JTable* definierte Methode. Sie erwirkt, dass wenn der Benutzer seine Maus auf eine Zelle bewegt, der Inhalt dieser Zelle als ToolTip angezeigt wird. Dies ist gerade dann sinnvoll, wenn die Zelle zu klein für ihren Inhalt ist.

- Die Methode *resizeColumnDisplay()* (siehe Listing 25) versucht jeder Spalte die kleinstmögliche Breite zu geben und den übrigen Platz auf die letzte Spalte aufzuteilen. Dies ist wenn wir 3-Address-Code darstellen wollen sehr sinnvoll, da in der letzten Spalte der Kommentar steht.

[6] Diese Funktionalität ist notwendig, da das *JTable*-Objekt sich als Listener hinzufügt.

2.3 Drei Address Code

Im folgenden Kapitel wird die Implementierung des drei Address Codes erläutert. Wie zu Beginn von Abschnitt 2 angeführt werden diese Klassen benötigt:

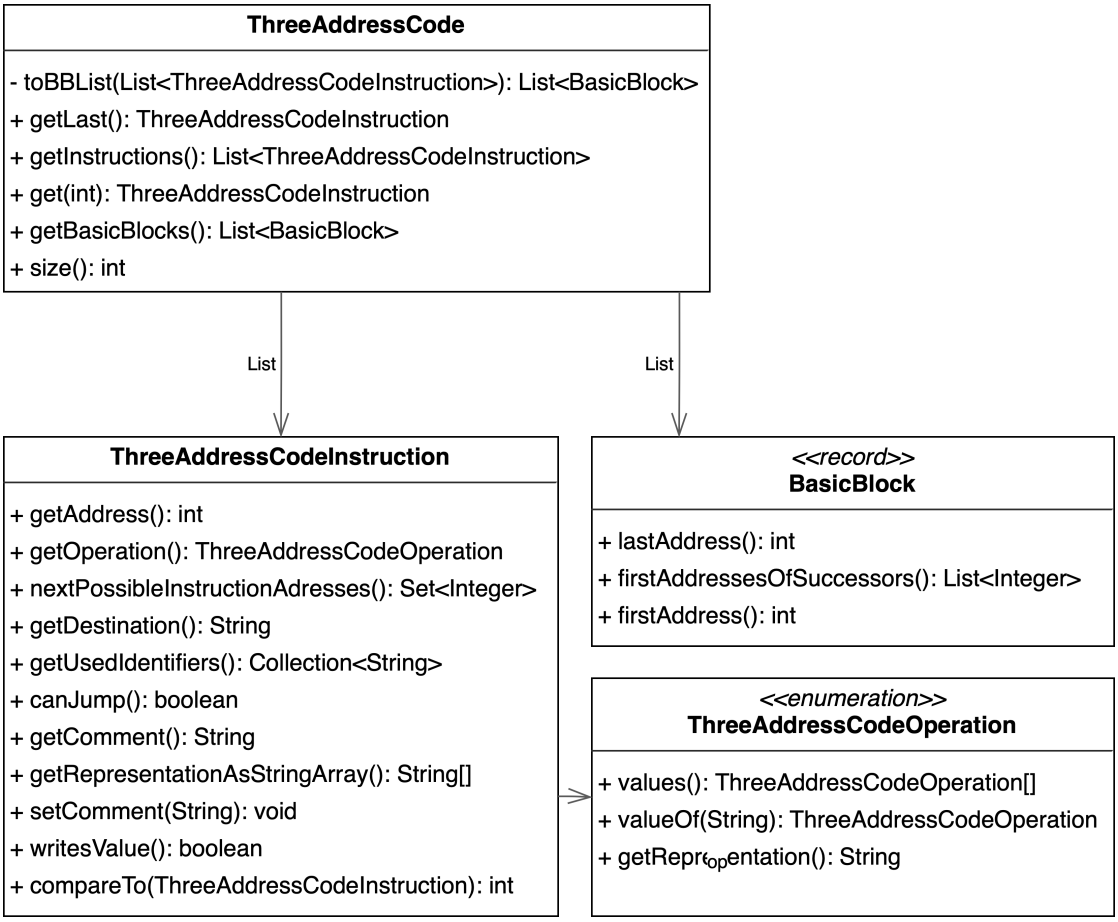

Abbildung 9: Implementation der drei Address Code Klassen

2.3.1 Die ThreeAddressCodeInstruction Klasse

Um einzelne Instruktionen zu modellieren wird die Klasse *ThreeAddressCodeInstruction* verwendet. Generiert werden diese aus einem String und einer Zahl, nämlich der Addresse an der Stelle ihres Programms. Dies ist zwar streng gesehen eine Ungenauigkeit der Modellierung, da jede Instruktion zwar eine Addresse hat, diese jedoch nicht in Bezug direkt auf die einzelne Instruktion steht, sondern nur im Kontext des Gesamtprogrammes gesehen werden kann.
Da es jedoch die Implementierung einiger folgender Methoden stark vereinfacht und keine Instruktion ohne eindeutige Addresse an der sie gespeichert oder ausgeführt werden kann existiert wurde sich dazu entschieden für diese ein Attribut anzulegen.

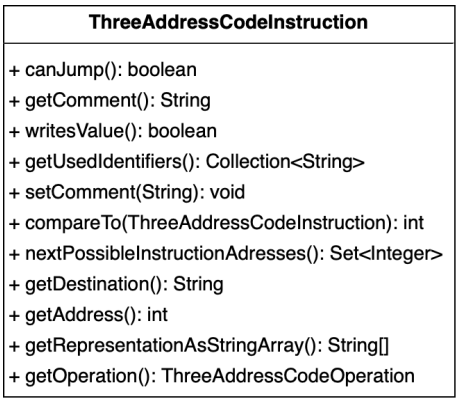

Abbildung 10: Die ThreeAddressCodeInstruction Klasse

Um die in Abschnitt 1.2.1 definierten Instruktionen zu modellieren werden folgende weitere Attribute benötigt:

1. Eine *ThreeAddressCodeOperation*. Dies ist ein Enum, welches definiert welche Operation die angegebene Instruktion ausführt.
2. Ein String namens *destination*. Dieser gibt entweder das Ziel eines Sprunges an, oder den Ort an dem eine Berechnung gespeichert werden soll.
3. Den String *source*. Da alle Instruktionen ausser dem unbedingten Sprung mindestens einen Wert verarbeiten.
4. Und einen String *modifier*, für Instruktionen die zwei Werte entweder vergleichen oder arithmetisch verrechnen.

Desweiteren wurde ein weiteres String-Attribut *comment* zum speichern von Kommentaren hinzugefügt.

Um aus dem Eingabestring ein passendes *ThreeAddressCodeInstruction*-Objekt zu generieren wird der Eingabestring im Konstruktor(Listing 26) in seine Bestandteile aufgelöst und der richtigen Instruktion zugeordnet.

Außerdem wurden folgende Helfermethoden, welche im späteren Verlauf dieser Arbeit benötigt werden, implementiert:

1. Die Methode *canJump()* gibt den Wahrheitswert *wahr* zurück, wenn die Instruktion springen kann oder immer springt, ansonsten gibt sie *falsch* zurück.
2. Die Methode *writesValue()* gibt den Wahrheitswert *wahr* zurück, wenn die Instruktion den Wert der in *destination* gespeichert ist überschreibt. Ansonsten gibt sie *falsch* zurück.
3. *getUsedIdentifiers()* gibt die Menge der Werte in *source* und *modifier* zurück, wenn diese Variablen referenzieren und keine Konstanten sind.
4. Die Methode *compareTo(ThreeAddressCodeInstruction)* implementiert das Interface Comparable<>. Wir sehen später dass dies sinnvoll ist wenn in Erfahrung gebracht werden soll sich ob zwei Instruktionen im selben Grundblock befinden.
5. *nextPossibleInstructionAddresses()* gibt die Menge der nächsten möglichen Instruktionsaddressen zurück. Dies ist in den meißten Fällen einfach die nächste Addresse, wenn die aktuelle Instruktion jedoch ein unbedingter Sprung ist, ist es der Wert in *destination*, wenn die Instruktion ein bedingter Sprung ist, wird sowohl die nächste Addresse als auch der Wert in *destination* zurückgegeben.
6. Die Methode *getRepresentationAsStringArray()* gibt eine Repräsenationen der Instruktion als *String[]* in der Form zurück, dass sie nacheinander die eingelesene Instruktion bilden zum Beispiel:

Die restlichen in *ThreeAddressCodeInstruction* enthaltenen Klassen sind getter und setter Methoden.

2.3.2 Die BasicBlock Record-Klasse

Um die Grundblöcke eines Programmes zu modellieren reicht uns eine Record-Klasse, da Grundblöcke immer ein gesamtes Programm partitionieren reicht es die erste und letzte der enthaltenen und die Menge der folgenden Addressen zu kennen.

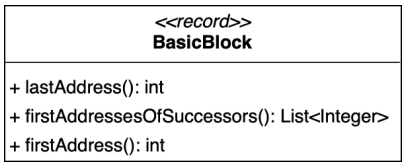

Abbildung 11: Die ThreeAddressCodeInstruction Klasse

2.3.3 Die ThreeAddressCode Klasse

Die Klasse *ThreeAddressCode* agiert im Framework als Modellierung eines gesamten Programmes. Sein Konstruktor(Listing 27) wird mit einem String aus 3-Address-Code Instruktionen welche durch einen Zeilenumbruch getrennt sind aufgerufen, daraus generiert er eine Liste and *ThreeAddressCodeInstruction* Objekten und aus diesen Objekten wiederum eine Liste an *BasicBlock* Objekten.

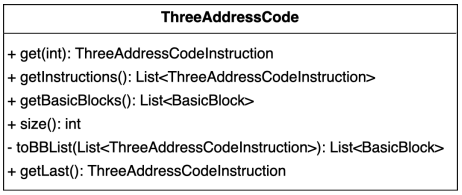

Abbildung 12: Die ThreeAddressCode Klasse

Die Methode *toBBList(List<ThreeAddressCodeInstruction>)* ist hier eine statische Methode, welche für eine Liste an *ThreeAddressCodeInstruction*-Objekten eine Liste an *BasicBlock*-Objekten zurückgibt. Dafür wurde der in Abschnitt 1.2.2 angegebene Algorithmus implementiert(Listing 28). In Folge dessen werden die nächstmöglichen Instruktionsaddressen bestimmt.

Um alle Leader zu finden wurde wie folgt über die Instruktionsliste iteriert(Listing 29). Wenn die Instruktion an der Stelle i springen kann, ist das Ziel des Sprungs ein Leader. Ausserdem ist die Instruktion an der Stelle i+1 auch ein Leader, wenn sie existiert Um die ersten Addressen der Nachfolger zu erhalten, schauen wir uns die letzte Addresse unseres Blockes an(Listing 30), da an dieser entweder entweder ein Sprung ist, an die nächste Addresse gesprungen wird oder das Programm zu ende ist. Mit diesen Klassen und Methoden wurden alle für die Implementation der Plugins notwendigen Modellierungen abgebildet.

2.4 Implementierung von Algorithmen

Dieser Abschnitt befasst sich damit wie Algorithmen als Plugins implementiert werden, sodass diese vom Framweork richtig visualisiert werden.

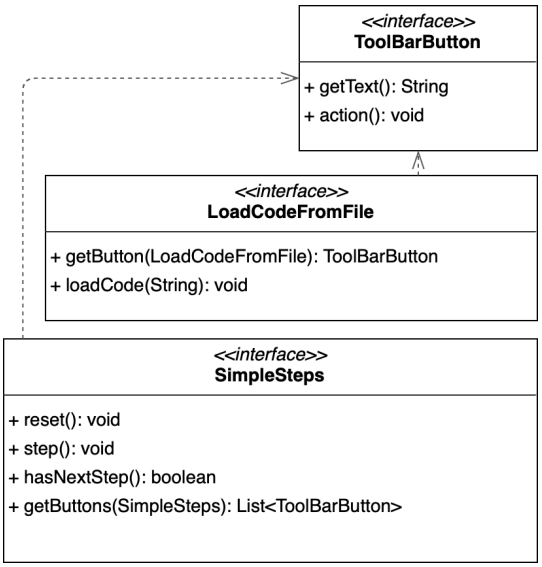

Abbildung 13: Das ToolBarButton Interface

Hierbei gibt es zwei Interfaces die Implementiert wurden, das *Plugin* Interface und das *ToolBarButton* Interface. Letzteres spezifiziert alle Buttons welche in der Toolbar angezeigt werden wenn das Plugin geladen ist. Es ist sehr einfach aufgebaut und verfügt nur über zwei Methoden. Eine Methode *getText()* gibt einen String zurück, welcher im Button angezeigt wird. Die andere Methode *action()* wird ausgeführt wenn der Nutzer auf den zugehörigen Button drückt.
Da alle in dieser Arbeit implementierten Algorithmen sowohl Schrittweise durchlaufen werden sollen als auch Code laden müssen wurden ausserdem zwei Interfaces implementiert welche ToolBarButtons für Plugins bereitstellen.

Das Interface *LoadCodeFromFile* stellt sicher dass es eine Funktion *loadCode(String)* gibt, in die Code als ein String in das Plugin geladen werden kann.
Die Funktion *getButton()* ist hier eine statische Funktion, welche eine Implementation von *ToolBarButton* zurückgibt welche zuerst einen *JFileChooser* [7] öffnet und dann die Datei als String lädt. Hierbei wird auch der Unterschied zwischen Windows und *nix basierenden Systemem berücksichtigt und die einzelnen Zeilen nur mit einem Zeilenumbruch(\n) konkateniert.

Das Interface *SimpleSteps* implementiert sogar drei Buttons und somit auch drei Methoden. Ein Reset Button, der die Methode *reset()* aufruft, ein Step Button, der die Methode *step()* aufruft und einen Run Button, der die Methode *step()* so lange aufruft wie die Methode *hasNextStep() wahr* zurückgibt. Diese drei Buttons werden wieder in einer statischen Funktion *getButtons()* generiert.

[7] *JFileChooser* ist eine Klasse aus der *swing*-Library welche eine Bedienfläche zum öffnen von Dateien bietet

2.4.1 Das Plugin Interface

Das *Plugin*-Interface (Abb. 14) beschreibt die Schnittstelle zwischen dem Framework und dem entwickelten Plugin. Um ein Plugin zu Implementieren benötigt es folgende Methoden:

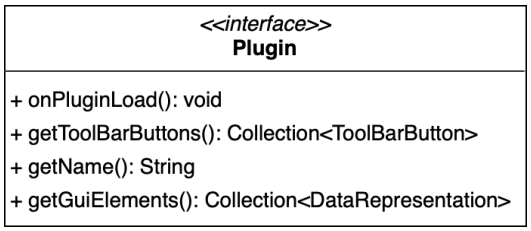

Abbildung 14: Das Plugin Interface

- *onPluginLoad()* wird ausgeführt wenn das Plugin geladen wird, also zum Beispiel wenn der Nutzer im Plugin Dropdown Menü auf den Button mit dem Namen des Plugins drückt.
- *getToolBarButtons()* gibt eine Collection der implementierten *ToolBarButtons* zurück, sodass diese wenn das Plugin geladen wird in die ToolBar eingefügt werden.
- *getName()* gibt den Namen des Plugins zurück, sodass er im Plugin Dropddown Menü angezeigt wird.
- *getGuiElements()* gibt eine Collection der *DataRepresentation* Elemente zurück die das Plugin anzeigen soll.

Mehr Abstraktionen braucht das Framework nicht um alle Algorithmen die in dieser Arbeit implementiert werden sollten zu Visualisieren. Im nächsten Kapitel wird nun auf die spezifische Implementationen der Algorithmen eingegangen.

3 Implementierte Algorithmen

Die folgenden Kapitel behandeln die Implementation der in Kapitel 1.2 vorgestellten Algorithmen.

Sämtliche Algorithmen wurden nach folgendem Muster implementiert: Es gibt eine Hauptklasse die das *Plugin*-Interface, das *SimpleSteps*-Interface und das *loadCodeFromFile*-Interface implementiert und eine zweite Klasse welche den Durchlauf eines Algorithmus Modelliert. Dies bedeutet, dass einige Methoden fast identisch implementiert wurden:

- *getName()* gibt den Namen des Plugins zurück.
- Die Methode *getToolBarButtons()* gibt eine Collection der Buttons zurück, die in den Statischen Methoden der jeweils implementierten Interfaces *SimpleSteps* und *loadCodeFromFile* generiert werden.
- *getGuiElements()* gibt eine Collection der Gui-Elemente zurück. Welche Gui-Elemente genau zurückgegeben werden, und an welcher Position diese sind unterscheidet sich allerdings von Plugin zu Plugin.
- Die *onPluginLoad()*-Methode(siehe Abb. 14) setzt das Plugin zurück indem es die *reset()*-Methode aufruft. Da alle Plugins *SimpleSteps*(siehe Abb. 13) implementieren, ist es für den Benutzer am intuitivsten, wenn der geladene Algorithmus sich zwar zurücksetzt, aber trotzdem den geladenen Code behält.
- *loadCode(String)* ersetzt den aktuell geladenen Code, welcher im Attribut *currentlyLoadedCode* gespeichert ist und führt dann auch die *reset()*-Methode aus.
- Die Methode *step()* führt eine gleichnamige Methode in der Algorithmus ausführenden Klasse auf und aktualisiert dann die Gui-Elemente indem es die Methode *refreshGuiElements()* ausführt.
- *hasNextStep()* gibt den negierten Wahrheitswert der Methode *isFinished()* der Algorithmusklasse zurück.
- Die *reset()* Methode generiert ein neues Objekt der zugehörigen Algorithmusklasse mit dem aktuell geladenen Code, setzt dann die Gui-Elemente zurück und initialisiert diese dann erneut.

Zudem hat jede *Plugin*-Klasse eine *refreshGuiElements()*-Methode in der das Gui aktualisiert wird. Diese ist nicht Teil des Interfaces, da das Framework nicht darauf zugreift. Da das Gui jedoch sowohl beim laden des Plugins, als auch beim Zurücksetzen und Durchschreiten des Algorithmusses aktualisiert werden muss, spart dies Codeduplizierungen ein.

3.1 Erstellung von Grundblöcken aus 3-Address-Code

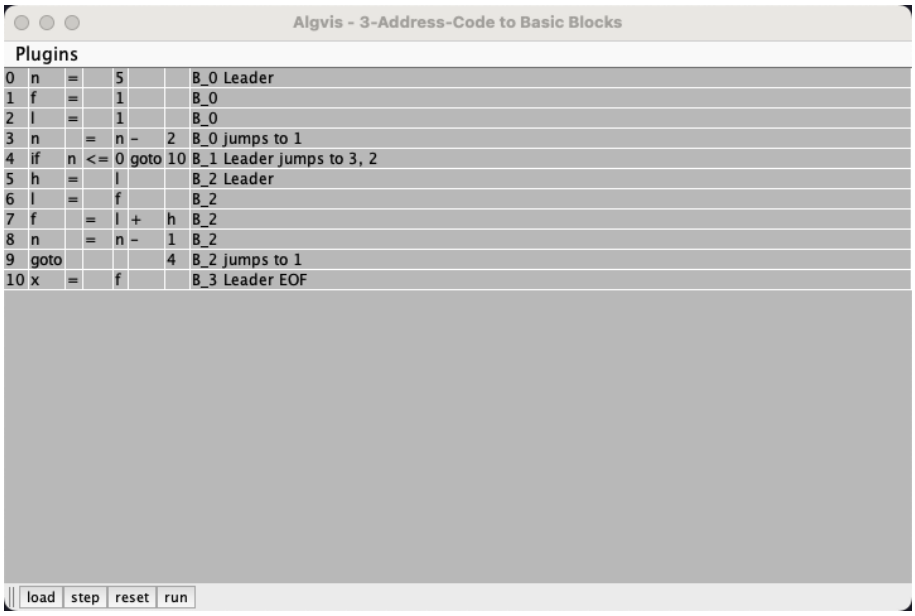

Abbildung 15: Das "3-Address-Code zu Grundblöcken"Plugin

Dieses Plugin implementiert den in Abschnitt 1.2.2 eingeführten Algorithmus und zeigt das 3-Address-Code Programm in einer Tabelle an (**??**). Nach der Ausführung des Programmes wird in der Kommentarspalte der Tabelle für jede Instruktion angezeigt zu welchem Grundblock sie gehört, zudem wird auch angezeigt welche die folgenden Grundblöcke für jeden einzelnen Grundblock sind.

Dafür wird beim Erstellen eines Objektes der Algorithmusklasse 3-Address-Code als String übergeben, ein *ThreeAddressCode*-Objekt und ein Set erstellt, in welchem die gefundene Leader gespeichert werden.

Durch das Aufrufen der *step()*-Methode wird nun zwei Mal durch die Instruktionen des *ThreeAddressCode*-Objekts iteriert.

Im ersten Durchlauf werden alle Leader markiert(Listing 3):

Listing 3: Markieren von Leadern

```
ThreeAddressCodeInstruction instruction = code.get(address);
if(instruction.canJump()) {
  ThreeAddressCodeInstruction destination = code.get(
    Integer.parseInt(instruction.getDestination())
  );
  leaders.add(destination);
  destination.setComment("Leader");
  if(address+1 < code.size()
  && instruction.getOperation() != ThreeAddressCodeOperation.jmp) {
    ThreeAddressCodeInstruction nextInstruction = code.get(address + 1);
    leaders.add(nextInstruction);
    nextInstruction.setComment("Leader");
  }
}
```

Im zweiten Durchlauf werden alle Instruktionen ihrem Block zugeordnet:

Listing 4: Zuordnen der Instruktionen zu ihren Blöcken

```
ThreeAddressCodeInstruction instruction = code.get(address);
if(leaders.contains(instruction)){
  int blockNumber = getSortedLeaders().indexOf(instruction);
  instruction.setComment("B_" + blockNumber + "␣Leader");
}else{
  ThreeAddressCodeInstruction lastLeader = getPreviousLeader(address);
  int blockNumber = getSortedLeaders().indexOf(instruction);
  instruction.setComment("B_" + blockNumber);
}

if(code.getLast().equals(instruction)) {
    ... // Kommentar wird mit Ziel des Sprungs ergaenzt
}else if(...) {
  ... // Dies wird im naechsten Plugin vollstaendig abgebildet
}

if(code.getLast().equals(instruction)) {
  instruction.setComment(instruction.getComment() + "␣EOF");
}
```

Hierbei ist die Methode *getSortedLeaders()* eine Hilfsmethode, welche das Set der erfassten Leader nach Addressen aufsteigend sortiert und als Liste wiedergibt.

Im Kommentar-String jeder einzelnen Instruktion ist nun angegeben in welchem Block sich diese befindet, welche Instruktion Leader dieses Blockes ist und zu welchen Blöcken die letzte Instruktion springt.
Um dies im Gui anzuzeigen wird nach jeder ausführung der *step()*-Methode die Methode *refreshGuiElements()*(Listing 5) in der Pluginklasse ausgeführt. diese aktualisiert alle Daten in der Tabelle und markiert dann die in diesem Schritt betrachtete Zeile.

Listing 5: Aktualisieren der Tabelle

```
for (int i = 0; i < pluginInstance.getCode().size(); i++) {
  code.setRowTo(
    pluginInstance.getCode()
      .get(i)
      .getRepresentationAsStringArray()
    , i);
}
code.highlightLine(pluginInstance.getCurrentInstructionAddress());
```

3.2 Erstellung eines Kontrollflussgraphen aus 3-Address-Code

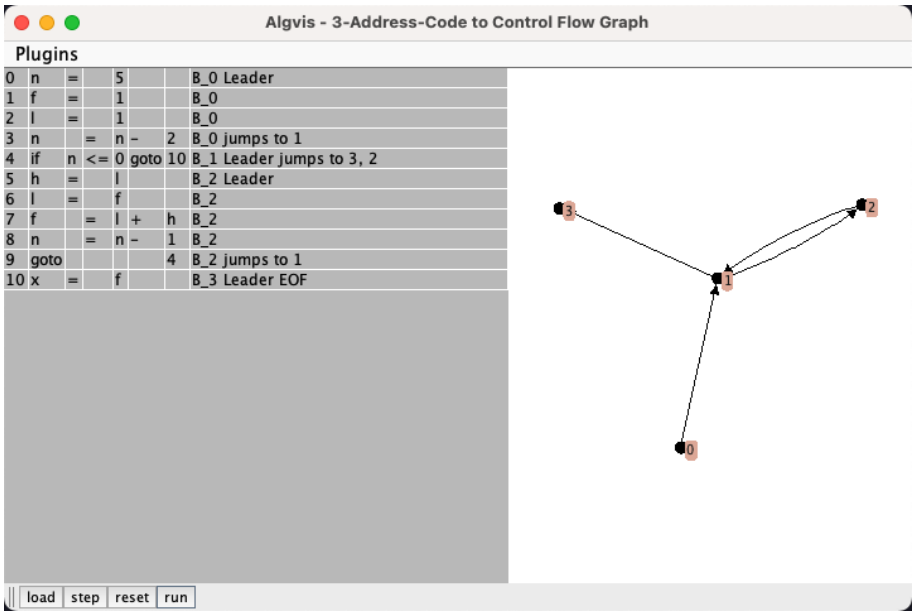

Abbildung 16: Das "3-Address-Code zu Grundblöcken"Plugin

Das Plugin zur Generierung eines Kontrollflussgraphen aus 3-Address-Code ist eine Erweiterung des Plugins zur Erstellung von Grundblöcken.

Es Erweitert das im letzten Abschnitt behandelte Programm indem es neben der Tabelle einen Kontrollflussgraphen anzeigt.

Um dies zu relalisieren wurden, neben einem Graphen, zwei Map-Objekte hinzugefügt. Eines in der Plugin-Klasse, um zu verfolgen welche Grundblöcke bereits einen Knoten im Graphen besitzen. Und eines in der Algorithmus-Klasse, um festzuhalten welche Grundblöcke auf andere folgen.

Die Methode zum Zuordnen der Instruktionen zu ihren Grundblöcken(Listing 4) wurde erweitert, sodass die Nachfolger der Grundblöcke nicht nur im Kommentar, sondern auch in der Map erfasst werden:

Listing 6: Erweiterung der *mapAddressesToItsBlock*-Methode

```
...
if(code.getLast().equals(instruction)) {
  if(instruction.canJump()){
    StringBuilder postfix = new StringBuilder("␣jumps␣to␣");
    ThreeAddressCodeInstruction nextInstruction =
      instruction.nextPossibleInstructionAdresses().stream()
      .map(code::get)
      .min(ThreeAddressCodeInstruction::compareTo)
      .get();
    postfix.append(getSortedLeaders().indexOf(nextInstruction));
    successorMap.put(getPreviousLeader(address), new HashSet<>(Set.of(nextInstru
    instruction.setComment(instruction.getComment() + postfix.toString());
  }
}else if(leaders.contains(code.get(address+1))) {
  StringBuilder postfix = new StringBuilder("␣jumps␣to␣");
  List<ThreeAddressCodeInstruction> nextInstructions =
    instruction.nextPossibleInstructionAdresses().stream()
    .map(code::get)
    .toList();
  for (int i = 0; i < nextInstructions.size(); i++) {
    postfix.append(getSortedLeaders().indexOf(nextInstructions.get(i)));
    if(i+1<nextInstructions.size())
      postfix.append(",␣");
  }
  successorMap.put(getPreviousLeader(address), new HashSet<>(nextInstructions));
  instruction.setComment(instruction.getComment() + postfix);
}

if(code.getLast().equals(instruction)) {
  instruction.setComment(instruction.getComment() + "␣EOF");
}
```

In den If Statements werden die Blöcke, in denen die Instruktionen die als nächstes ausgeführt werden, ermittelt und an das Ende des Kommentars gehangen. Ausserdem werden die Addressen der nächsten Instruktionen in die *successorMap* zugefügt [8].

[8] Im Plugin „3-Address-Code to Basic Block “existiert diese map nicht, daher hat die Methode dort die beiden Befehle auch nicht

Die *refreshGuiElements()*-Methode wurde auch um folgende Zeilen erweitert:

Listing 7: Aktualisieren der Tabelle

```
//set CFG
List<ThreeAddressCodeInstruction> leaders =
  currentPluginInstance.getSortedLeaders();
for (ThreeAddressCodeInstruction leader :
  currentPluginInstance.getSortedLeaders()) {
  //add new Nodes
  if(!nodeMap.containsKey(leader)) {
    Node node = new Node();
    nodeMap.put(leader, node);
    controlFlowGraph.addNode(nodeMap.get(leader));
  }
  controlFlowGraph.setLabelOfNode(
    nodeMap.get(leader), Integer.toString(leaders.indexOf(leader))
  );
}
//add all edges
Map<ThreeAddressCodeInstruction
  , Set<ThreeAddressCodeInstruction>> successorList =
    currentPluginInstance.getSuccessorMap();
for (ThreeAddressCodeInstruction source:successorList.keySet()) {
  Node sourceNode = nodeMap.get(source);
  for (ThreeAddressCodeInstruction destination:
    successorList.get(source)) {
    Node destinationNode = nodeMap.get(destination);
    controlFlowGraph.addEdge(new Edge(sourceNode, destinationNode));
  }
}
```

In der ersten Schleife wird für jeden (bis zu diesem Schritt) gefundenen Leader ein Knoten im Graphen erstellt wenn dieser noch nicht hinzugefügt worden ist und das Label jedes Leaders aktualisiert, da nicht zu jedem Zeitpunkt jeder Leader gefunden kann nicht davon ausgegangen werden dass die Label aus dem letzten Schritt noch aktuell sind.
In der zweiten Schleife werden für alle Leader Kanten zu den erkannten Nachfolgern gezeichnet.

3.3 Analyse von erreichenden Definitionen

Das Plugin zur Visualisierung der Analyse der erreichenden Definitionen(Abb. 17) hat eine weitere Tabelle, in der Datenflusswerte angezeigt werden.

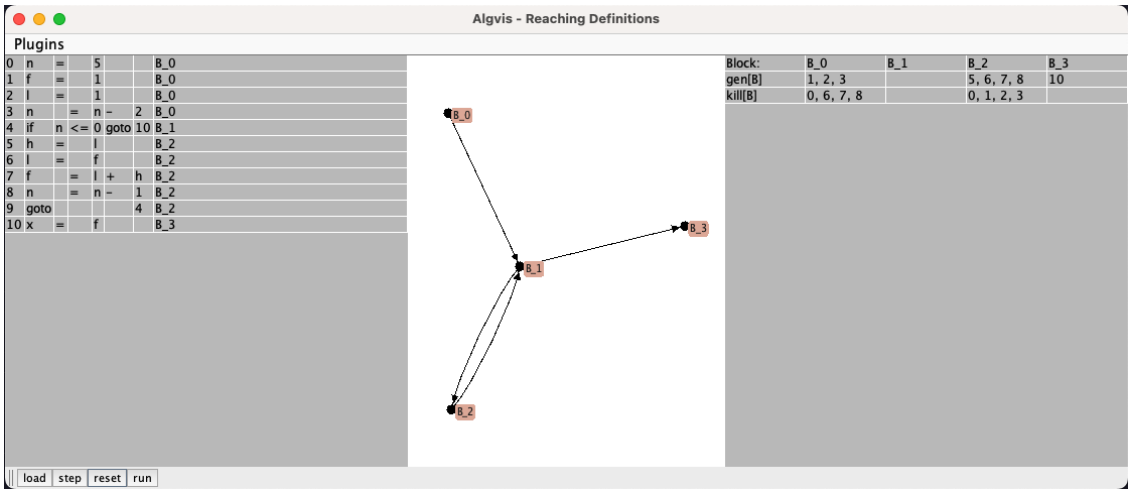

Abbildung 17: Das "Reaching Definitions"Plugin

Da sich sowohl der Kontrollflussgraph, als auch der 3-Address-Code im Verlaufe des Programmes nicht verändern, werden diese nicht in der *refreshGuiElements()*-Methode behandelt, sondern in der *reset()*-Methode. da in dieser ein neues Objekt der Algorithmusklasse erstellt wird und sich der Code nur dort ändern kann.

In der *refreshGuiElements()*-Methode verändert sich dementsprechend nur die Tabelle in der Datenflusswerte angezeigt werden.

Diese berechnen wir, wie in Abschnitt 1.2.4 erläuftert, in der Algorithmusklasse:

Die gen_B und $kill_B$-Mengen berechnen wir als Maps für jeden Grundblock B:

Listing 8: Berechnung einer gen und $kill$-Menge

```
//gen
Set<Integer> currentGenSet = gen.get(block);
for (int i = block.lastAddress(); i >= block.firstAddress(); i--) {
  ThreeAddressCodeInstruction currentInstruction = code.get(i);
  if(!currentInstruction.writesValue())
    continue;
  boolean newVar = true;
  for (int j : currentGenSet) {
    if(currentInstruction.getDestination().equals(code.get(j).getDestination()))
      newVar = false;
```

```
    }
    if(newVar)
      currentGenSet.add(i);
  }
  //kill
  Set<Integer> currentKillSet = kill.get(block);
  for (Integer i:currentGenSet) {
    for (int j = 0; j < code.size(); j++) {
      if(!code.get(j).writesValue())
        continue;
      if(j == i)
        continue;
      if(code.get(j).getDestination().equals(code.get(i).getDestination()))
        currentKillSet.add(j);
    }
  }
```

Beim Durchlaufen des Algorithmus betrachten berechnen wir in einem Schritt die $in[B]$ und $out[B]$ Mengen für einen Grundblock B. Wenn diese sich von der in der letzten Iteration berechneten Menge unterscheidet ist unser Fixpunkt noch nicht erreicht.

Da wir nicht nur die aktuellen in und out-Mengen darstellen wollen, sondern alle, werden diese in einer Liste gespeichert.

Die $in_i[B]$-Menge wird als Vereinigung aller $out[B]$-Mengen der Vorgänger des Blocks B berechnet(**??**):

Listing 9: Berechnen der in-Menge

```
Set<String> currentIn =  new HashSet<>();
for (BasicBlock ancestorBlock:ancestorBlocks) {
  Set<String> lastOut = (currentIteration>0)
    ? out.get(currentIteration-1).get(ancestorBlock)
    : Set.of(); //Leeres Set wenn es kein vorherige Iteration gibt
  currentIn.addAll(lastOut);
}
```

Die gen_B und $kill_B$ Mengen wurden auf den Addressen der Instruktionen definiert, werden jedoch als Addressen benötigt, daher werden sie wie folgt konvertiert:

Listing 10: Konvertierung der gen-Menge

```
List<String> generatedIdentifiers =
  gen.get(currentBlock).stream()
    .map(j -> code.get(j).getDestination())
    .toList();
```

```
List<String> killedIdentifiers =
  kill.get(currentBlock).stream()
    .map(j -> code.get(j)
    .getDestination())
    .toList();
```

Um nun die $out[B]$-Menge zu berechen werden alle Elemente der $kill_B$-Menge aus einer Kopie der $in[B]$-Menge entfernt. Anschließend fügen wie alle Elemente der gen_B-Menge hinzu:

Listing 11: Berechnung der out-Menge

```
Set<String> currentOut = new HashSet<>(currentIn);
currentOut.removeAll(killedIdentifiers);
currentOut.addAll(generatedIdentifiers);
```

Um die berechneten Datenflusswerte anzuzeigen wird die Tabelle wie im Beispiel(Tabelle 1)befüllt:

Die erste Zeile dient als Überschriftszeile für die Grundblöcke. Die zweite und dritte Zeile befüllen die gen_B und $kill_B$ Mengen(Listing 12):

Listing 12: Berechnung der out-Menge

```
dataFlow.setValueAt("gen[B]", 1, 0);
dataFlow.setValueAt("kill[B]", 2, 0);
for (int col = 0; col < basicBlocks.size(); col++) {
  BasicBlock block = basicBlocks.get(col);
  String genBlock = collectStringSet(
    pluginInstance.getGen().get(block).stream()
      .map(String::valueOf)
      .collect(Collectors.toSet())
    );
  String killBlock = collectStringSet(
    pluginInstance.getKill().get(block).stream()
      .map(String::valueOf)
      .collect(Collectors.toSet())
    );
  dataFlow.setValueAt(genBlock, 1, col+1);
  dataFlow.setValueAt(killBlock, 2, col+1);
}
```

Die Methode *collectStringSet* ist hier eine Helfermethode, welche ein Set an Strings zu einem String kombiniert in dem die einzelnen Elemente durch Kommas getrennt werden.

In den folgenden Zeilen werden die $in[B]$ und $out[B]$ Mengen aufgelistet. da in der letzten

Iteration eventuell noch nicht alle Mengen bestimmt worden sind wird für die unbestimmten Mengen ein Platzhalter-Set mit dem Element „-“eingefügt:

Listing 13: Visualisierung der in und out-Mengen

```
for (int i = 0; i < inTable.size();i++) {
  int rowIndex = (i*2)+3;
  Map<BasicBlock, Set<String>> inMap = inTable.get(i);
  Map<BasicBlock, Set<String>> outMap = outTable.get(i);
  dataFlow.setValueAt("in[B]^"+i, rowIndex, 0);
  dataFlow.setValueAt("out[B]^"+i, rowIndex+1, 0);
  for (int j = 0; j < basicBlocks.size(); j++) {
    Set<String> inBlock = inMap.getOrDefault(basicBlocks.get(j), Set.of("-")
    String inString = collectStringSet(inBlock);
    dataFlow.setValueAt(inString, rowIndex, j+1);

    Set<String> outBlock = outMap.getOrDefault(basicBlocks.get(j), Set.of("-
    String outString = collectStringSet(outBlock);
    dataFlow.setValueAt(outString, rowIndex+1, j+1);
  }
}
```

3.4 Analyse von lebendigen Variablen bezüglich Grundblöcken

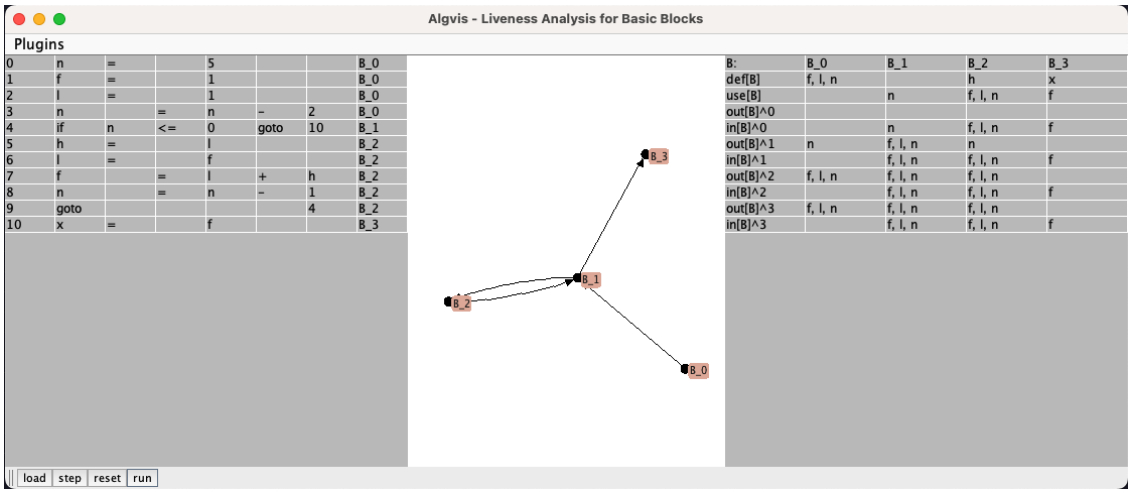

Abbildung 18: Das "Liveness Analysis for Basic Blocks"Plugin

Im folgenden Kapitel wird die Implementation des Algorithmus für die Liveness Analysis besprochen. Da der Aufbau der Gui identisch zu dem Plugin für errichende Definitionen ist, wird darauf nicht erneut eingegangen.

Da die Datenflussrichtung rückwärts läuft, wird die $out[B]$-Menge wie folgt bestimmt:

Listing 14: Berechnung der out-Menge

```
Set<String> currentOut = new HashSet<>();
for (BasicBlock successor:successorBlocks) {
  Set<String> lastInSuccessor = (currentIteration>0)
    ? in.get(currentIteration-1).get(successor)
    : Set.of();
  currentOut.addAll(lastInSuccessor);
}
```

Um den Algorithmus wie in Abschnitt 1.2.5 zu implementieren müssen für alle Grundblöcke B die def_B und use_B Mengen definiert werden. Dies geschieht wie folgt(Listing 15):

Listing 15: Berechnung der def und use-Mengen

```
//def
Set<String> currentDefSet = def.get(block);
for (int i = block.lastAddress(); i >= block.firstAddress(); i--) {
  ThreeAddressCodeInstruction instruction = code.get(i);
  if(!instruction.writesValue())
    continue;
  String identifier = instruction.getDestination();
  currentDefSet.add(identifier);
  for (int j = 0; j < i; j++) {
    if(code.get(j).getUsedIdentifiers().contains(identifier))
      currentDefSet.remove(identifier);
  }
}
//use
Set<String> currentUseSet = use.get(block);
for (int i = block.lastAddress(); i >= block.firstAddress(); i--) {
  ThreeAddressCodeInstruction instruction = code.get(i);
  for (String used:instruction.getUsedIdentifiers()) {
    currentUseSet.add(used);
    for (int j = block.firstAddress(); j < i; j++) {
      if(code.get(j).writesValue() && code.get(j).getDestination().equals(us
        currentUseSet.remove(used);
    }
  }
}
```

Und die Transferfunktion wird wie folgt berechnet:

Listing 16: Berechnung der out-Menge

```
Set<String> currentIn = new HashSet<>(currentOut);
currentIn.removeAll(def.get(currentBlock));
currentIn.addAll(use.get(currentBlock));
```

3.5 Analyse von lebendigen Variablen bezüglich einzelnen Instruktionen

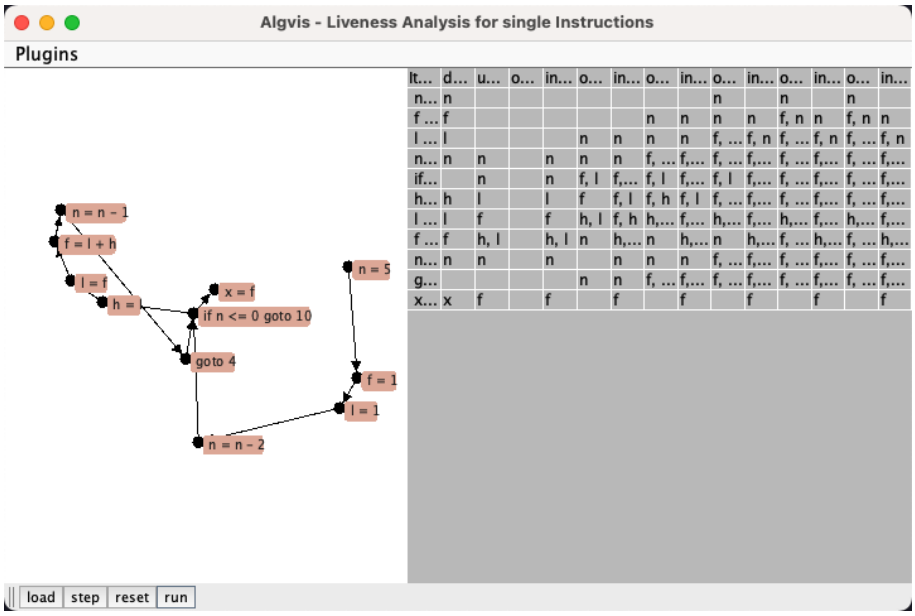

Abbildung 19: Das "Liveness Analysis for Three Address Code"Plugin

Durch die im letzten Abschnitt definierte Liveness Analyse lässt sich mit wenigen Iterationen eine Konservative Abschätzung der benötigten Register machen. Da sich jedoch auch innerhalb eines Grundblockes die Werte der Liveness Analyse ändern können, kann die liveness Analyse auch auf einzelnen Instruktionen geführt werden, hierbei werden jedoch wesentlich mehr Iterationen benötigt bis der Fixpunkt erreicht ist.

Dieses Plugin(Abb. 19) implementiert eben diese Analyse. Indem es alle Instruktionen in einem Graphen anzeigt und deren Datenflusswerte in einer Tabelle.

Die Transferfunktion bleibt identisch zur liveness Analyse auf Grundblöcken. Jedoch müssen die use_i und def_i-Mengen auf einzlnen Instruktionen i definiert werden:

- Die use_i-Menge umfasst trivialer Weise alle Addressen die in ihrer Instruktion i benötigt werden.
- Die def_i-Menge ist definiert durch: Alle Instruktionen die einen Wert definieren haben ein Element in ihrer def_i Menge, nämlich die Addresse in der dieser Wert gespeichert wird. Wenn eine Instruktion keinen Wert speichert, gilt $def_i = \emptyset$.

4 Verwandte Arbeiten

Wie im Abschnitt 1.1 bereits erwähnt, gibt es einige andere Arbeiten, welche ähnliche Aufgaben haben:

Eine davon ist die STUPSToolbox[FR] programmiert von Fabian Ruhland [9] und erweitert von Isabel Wingen.
Diese Toolbox enthält einige Funktionaliäten für die Arbeit mit Automaten und Grammatiken. Ursprünglich war die Idee diese Toolbox weiter zu entwickeln, jedoch funktioniert diese leider mittlerweile nicht mehr ohne weiteres. Daher wurde sich dagegen entschieden diese Toolbox zu erweitern oder instand zu setzen.

Ein Tool mit dem auch liveness Analyse gemacht werden kann ist der Data Flow Analysis Visualizer[Kasa]. Dies ist eine Abschlussarbeit[Kasb] eines Compilerdesign Kurses der UCSD [10]. Leider kann in diesem Programm nur JavaScript-Code eingegeben werden und auch nur eine Liveness Analyse auf einzelnen Instruktionen ausgeführt werden.

Letztlich gibt es noch ein Tool namens VisOpt[fra]. In diesem kann man Jova[11]-Code eingeben, ihn in 3-Address-Code übersetzen lassen und sich eine Vielzahl von Optimierungsmöglichkeiten visualisieren lassen.
Leider ist es jedoch nicht möglich direkt 3-Address-Code einzugeben, sich Grundblöcke anzeigen zu lassen oder eine reaching Definitions Analyse durchzuführen.

[9] Warum kommt mir der Name so bekannt vor? :)
[10] University of California San Diego
[11] Eine Subsprache von Java

5 Ausblick

Das in dieser Arbeit entwickelte Framework bietet ein Fundament um viele Analyse- und Optimierungsalgorithmen zu visualisieren.
Leider ist es im vorgegebenen Zeitrahmen nicht möglich gewesen mehr als die in der Arbeit besprochenen Plugins zu schreiben.

Folgende Algorithmen können ohne Erweiterung des Frameworks implementiert werden:

- Konstantenpropagation
- Verfügbare Ausdrücke
- Eliminierung von Redundanz

Durch Implementierung von Automaten im Framework können ausserdem weitere Konzepte visualisiert werden die Studenten des Compilerbaus helfen könnten:

- Erstellen von Lexern für die Tokenisierung
  1. nicht-deterministischer endlicher Automat (NEA)
  2. deterministischer endlicher Automat (DEA)
  3. minimierter DEA

  Ausserdem auch durchführung einer Tokenisierung von Quellcode
- Erstellen von Parsern für den Bau von Syntaxbäumen
  1. Top-Down parsing
  2. Bottom-Up parsing
- Generierung von 3-Address-Code aus einem Syntaxbaum

6 Herausforderungen und Fazit

Das im Exposee vereinbarte Ziel dieser Bachelorarbeit war es folgende Konzepte zu visualisieren:

- Das Bauen eines Kontrollflussgraphen aus 3-Addres-Code
- Das Visualisieren von Constant Folding
- Das Visualisieren von liveness Analysen
- Und das Visualisieren von reaching Definitions

Leider war es aufgrund von falsch eingeschätztem Arbeitsaufwand nicht möglich den constant Folding Algorithmus zu implementieren.

Das Implementieren des Frameworks war hingegen sehr erfolgreich.
Es war möglich ein minimales Interface zu erstellen, welches alle Funktionen des Frameworks abbilden kann. Mit diesem können neue Plugins mit sehr geringem Aufwand hinzugefügt werden.

Ausserdem wurde das Framework samt Plugins einer Gruppe Student*innen des Compilerbau Moduls vorgestellt. Das Feedback der Studierenden war positiv, die vorgestellten Plugins seien nützlich um zum Beispiel Übungsaufgaben zu bearbeiten, da die Lösung Schritt für Schritt abgleichbar ist.

Anhang

Anhang A Codebeispiele

Listing 17: Generieren der ID für eine Kante

```
public String getID(){
  return "("+sourceNode.getId()+") -> ("+targetNode.getId()+")";
}
```

Listing 18: Konstruktor der Klasse ThreeAddressCodeInstruction

```
public Graph(Location location){
  this.location = location;
  exportedPanel = new JPanel(new BorderLayout());
  switch(location){...} // Groesse des exportierten JPanels festlegen

  nodes = new HashSet<>();
  edges = new HashSet<>();

  graph = new MultiGraph("Graph");
  graph.setAttribute("ui.stylesheet", ...); // stylesheet festlegen
  viewer = new SwingViewer(graph, ...); // threading festlegen
  View view = viewer.addDefaultView(false);

  layout = new LinLog();
  viewer.enableAutoLayout(layout);

  exportedPanel.add((Component) view, BorderLayout.CENTER);
}
```

Listing 19: Implementierung der addNode(Node) Methode

```
  public void addNode(Node newNode){
    if(nodes.contains(newNode))
      return;
    nodes.add(newNode);
    graph.addNode(newNode.getId());
    layout.shake();
  }
```

Listing 20: Implementierung der setLabelOfNode(Node, String) Methode

```
  public void setLabelOfNode(Node Node, String label){
    if(nodes.contains(newNode))
```

```
        graph.getNode(node).setAttribute("ui.label", label);
    }
```

Listing 21: Implementierung der setValueAtMethode(Object, int, int) Methode

```
    public void setValueAt(Object value, int rowIndex, int colIndex){
        if(value == null)
            value = "";
        try{
            data[rowIndex][colIndex] = value.toString();
            TableModelEvent event = new TableModelEvent(...);
            listeners.forEach(l->l.tableChanged(event));
        }catch(ArrayIndexException e){
            // Error handling
        }
    }
```

Listing 22: Implementierung der resizeTable(int, int) Methode

```
public void resizeTable(int rows, int cols) {
    if(rows==tableModel.getRowCount()
    && cols == tableModel.getColumnCount())
        return;
    tableModel = new DataTableModel(rows, cols);
    this.setModel(tableModel);
}
```

Listing 23: Implementierung der highlightLine(int) Methode

```
public void highlightLine(int line) {
    this.clearSelection();
    if (line == 0)
        return;
    try {
        this.addRowSelectionInterval(line, line);
    }catch (IllegalArgumentException e){
        ... //Handle wrong lines error
    }
        }
}
```

Listing 24: Implementierung der getToolTipText(MouseEvent) Methode

```
    public String getToolTipText(MouseEvent mouseEvent) {
        String tip = null;
        Point p = mouseEvent.getPoint();
```

```
    int rowIndex = rowAtPoint(p);
    int colIndex = columnAtPoint(p);

    try{
      if(rowIndex<getRowCount() && rowIndex> -1
      && colIndex<getColumnCount() && colIndex> -1)
        tip = tableModel.getValueAt(rowIndex, colIndex).toString();
    }catch (NullPointerException ignored){}

    return tip;
  }
```

Listing 25: Implementierung der resizeColumnDisplay() Methode

```
  public void resizeColumnDisplay() {
    TableColumnModel columnModel = this.getColumnModel();
    for (int i = 0; i < columnModel.getColumnCount()-1; i++) {
      int width = 0;
      for (int j = 0; j < tableModel.getRowCount(); j++) {
        TableCellRenderer renderer = this.getCellRenderer(j, i);
        Component component = this.prepareRenderer(renderer, j, i);
        width = Math.max(component.getPreferredSize().width+2, width);
      }
      columnModel.getColumn(i).setMinWidth(width);
      columnModel.getColumn(i).setMaxWidth(width);
    }
    int width = 10;
    for (int j = 0; j < tableModel.getRowCount(); j++) {
      TableCellRenderer renderer = this.getCellRenderer(j, columnModel.getColumn
      Component component = this.prepareRenderer(renderer, j, columnModel.getColu
      width = Math.max(component.getPreferredSize().width+2, width);
    }
    columnModel.getColumn(columnModel.getColumnCount()-1).setMinWidth(width);
    columnModel.getColumn(columnModel.getColumnCount()-1).setMaxWidth(this.getMa
  }
```

Listing 26: Konstruktor der Klasse ThreeAddressCodeInstruction

```
public ThreeAddressCodeInstruction(String rawInput, int address){
  ...
  String[] pieces = rawInput.split("␣");
  switch (pieces.length) {
    case 2 -> ... //Unbedingter Sprung goto X
    case 3 -> ... //Kopierbefehl X = Y
    case 4 -> { //entweder unaere Operation oder ein bedingter Sprung
```

```
      switch(pieces[2]){
        case "-" -> ... //Unaere Operation X = - Y
        case "goto" -> {//ein bedingter Sprung
          switch(pieces[0]){
            case "if" -> ... //bedingter Sprung if Y goto X
            case "ifFalse" -> ... //bedingter Sprung ifFalse Y goto X
          }
        }
      }
    case 5 -> ... //Binaere Operation X = Y op Z
    case 6 -> { //bedingter Sprung
      switch(pieces[2]){
        case "<" -> ...//if Y < Z goto X
        case ">" -> ...//if Y > Z goto X
        case "<=" -> ...//if Y <= Z goto X
        case ">=" -> ...//if Y >= Z goto X
        case "==" -> ...//if Y == Z goto X
        case "!=" -> ...//if Y != Z goto X
      }
    }
    ...
 }
```

Listing 27: Der Konstruktor der ThreeAddressCode Klasse

```
public ThreeAddressCode(String raw){
  List<String> inputLines = raw.lines().toList();
  code = new ArrayList<>(inputLines.size());
  for (int i = 0; i < inputLines.size(); i++) {
    code.add(new ThreeAddressCodeInstruction(inputLines.get(i), i));
  }
  basicBlocks = toBBList(code);
}
```

Listing 28: Zusammenfassung der toBBList() Methode

```
private static List<BasicBlock> toBBList(List<...> code){
  Set<ThreeAddressCodeInstruction> leaders = new HashSet<>(1);
  if(!code.isEmpty()) //die erste Instruktion ist immer Leader
    leaders.add(code.get(0));
  for (int i = 0; i < code.size(); i++) {
    ... // Finde alle Leader
  }
  List<ThreeAddressCodeInstruction> sortedLeaders = leaders.stream().sorted(
```

```
  // Finde die letzte Instruktion jedes Grundblockes
  List<Integer> firstAddresses = sortedLeaders.stream()
                .map(ThreeAddressCodeInstruction::getAddress)
                .toList();
  List<Integer> lastAddresses = new ArrayList<>(leaders.size());
  for (int i = 1; i < sortedLeaders.size(); i++) {
    lastAddresses.add(firstAddresses.get(i)-1);
  }
  lastAddresses.add(code.getLast().getAddress());

  List<List<Integer>> fAOS = new ArrayList<>(leaders.size());
  for (int lastAddress : lastAddresses) {
    ... // Bestimme die Addressen der Nachfolger
  }


  // Setze die Listen zu einer BasicBlock Liste zusammen
  List<BasicBlock> basicBlocks= new ArrayList<>(leaders.size());
  for (int i = 0; i < firstAddresses.size(); i++) {
    int firstAddress = firstAddresses.get(i);
    int lastAddress = lastAddresses.get(i);
    List<Integer> successors = fAOS.get(i);
    BasicBlock basicBlock = new BasicBlock(
      firstAddress, lastAddress, successors
    );
    basicBlocks.add(basicBlock);
  }
  return basicBlocks;
}
```

Listing 29: Ausschnitt der toBBList()-Methode zum finden aller Leader

```
for (int i = 0; i < code.size(); i++) {
  if(code.get(i).canJump()){
    ThreeAddressCodeInstruction destination = code.get(
      Integer.parseInt(code.get(i).getDestination())
    );
    leaders.add(destination);
    if(i+1<code.size())
      leaders.add(code.get(i+1));
} }
```

Listing 30: Zusammenfassung der toBBList() Methode

```
List<List<Integer>> fAOS = new ArrayList<>(leaders.size());
for (int lastAddress : lastAddresses) {
  List<Integer> successors = new ArrayList<>(0);
  fAOS.add(successors);
  switch (code.get(lastAddress).getOperation()) {
    case jmp ->
      successors.add(
        Integer.valueOf(code.get(lastAddress).getDestination())
      );
    case booleanJump, negatedBooleanJump,
         eqJump, geJump, gtJump, leJump, ltJump, neJump -> {
      successors.add(
        Integer.valueOf(code.get(lastAddress).getDestination())
      );
      if (lastAddress+1 < code.size())
        successors.add(lastAddress + 1);
    }
    default -> {
      if (lastAddress+1 < code.size())
        successors.add(lastAddress + 1);
} } }
```

Listing 31: Bestimmmen der gen Menge für erreichende Definitionen

```
for(BasicBlock block:basicBlocks){
  Set<Integer> currentGenSet = gen.get(block);
  for (int i = block.lastAddress(); i >= block.firstAddress(); i--) {
    ThreeAddressCodeInstruction currentInstruction = code.get(i);
    if(!currentInstruction.writesValue())
      continue;
    boolean newVar = true;
    for (int j : currentGenSet) {
```

```
      if(currentInstruction.getDestination().equals(code.get(j).getDestination()
        newVar = false;
    }
    if(newVar)
      currentGenSet.add(i);
  }
}
```

Listing 32: Bestimmmen der kill Menge für erreichende Definitionen

```
for(BasicBlock block:basicBlocks){
  Set<Integer> currentKillSet = kill.get(block);
  for (Integer i:currentGenSet) {
    for (int j = 0; j < code.size(); j++) {
      if(!code.get(j).writesValue())
        continue;
      if(j == i)
        continue;
      if(code.get(j).getDestination().equals(code.get(i).getDestination()))
        currentKillSet.add(j);
    }
  }
}
```

Listing 33: Bestimmmen der Menge der Vorfahren für erreichende Definitionen

```
for (BasicBlock block:basicBlocks) {
  List<Integer> firstAddressesOfSuccessors = block.firstAddressesOfSuccessors();
  for (int address:firstAddressesOfSuccessors) {
    for (BasicBlock potentialSuccessor:basicBlocks) {
      if(potentialSuccessor.firstAddress() == address)
        ancestors.get(potentialSuccessor).add(block);
    }
  }
}
```

Abbildungsverzeichnis

Tabellenverzeichnis

Literatur

[Aho08] AHO, Alfred V. ; LEUSCHEL, Michael (Hrsg.): *Compiler Prinzipien, Techniken und Werkzeuge*. 2., aktualisierte Aufl., German language ed. München u.a. : Pearson Studium, 2008 (It, Informatik). https://digitale-objekte.hbz-nrw.de/storage/2008/03/03/file_131/2343468.pdf

[FR] FABIAN RUHLAND, Isabel W.: *STUPSToolbox*. https://github.com/isabelwingen/STUPS-Toolbox-2.0

[fra] FRANZMANDL: *VisOpt: Visual Optimizer*. https://github.com/franzmandl/visopt

[GTa] GRAPHSTREAM-TEAM: *Graph Implementations*. https://graphstream-project.org/doc/FAQ/The-Graph-Class/What-are-the-Graph-implementations/. – zuletzt besucht am 30.01.2025

[GTb] GRAPHSTREAM-TEAM: *Graph Visualisation*. https://graphstream-project.org/doc/Tutorials/Graph-Visualisation/. – zuletzt besucht am 30.01.2025

[GTc] GRAPHSTREAM-TEAM: *GraphStream*. https://graphstream-project.org/. – zuletzt besucht am 30.01.2025

[GTd] GRAPHSTREAM-TEAM: *Storing, retrieving and displaying data in graphs*. https://graphstream-project.org/doc/Tutorials/Storing-retrieving-and-displaying-data-in-graphs/. – zuletzt besucht am 31.01.2025

[Kasa] KASHIWA, Shun: *Data-Flow Analysis Visualizer Sourcecode*. https://github.com/shumbo/DFAV

[Kasb] KASHIWA, Shun: *Data-Flow Analysis Visualizer Vorstellung*. https://sorensenucsc.github.io/CSE211-fa2021/projects/data-flow-analysis-visualizer/