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

Die Ziele dieser Bachelorarbeit waren einerseits einige Algorithmen und Konzepte aus dem Compilerbau zu visualisieren um Personen, insbesondere Studenten, eine Hilfe beim studieren dieser zu geben.

Zudem 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 einen 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 starkt erleichtert.

In dieser Bachelorarbeit wird ein Framework in Java entwickelt, mit dem es einfach sein soll Algorithmen zu visualisieren. 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 Algorihmen 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(beziehungsweise Variablen oder Konstanten) zugreifen, also die Variable in die der resultierende Wert gespeichert wird und eine oder zwei Variablen 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 Y 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
   In denen wenn der Wert Y entweder true oder false 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 Implementierte Relationen sind: Y $<$ Z, Y $\leq$ Z, Y $>$ Z, Y $\geq$ Z, Y $=$ Z und 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 umfangreichen 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 be
klären, dass jem
nix mit coba am
das auch verste

Grundblöcke sind Instruktionsfolgen eines Zwischencodeprogrammes in denen diese 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:

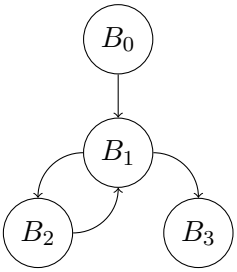

Abbildung 1: Resultierender Kontrollflussgraph

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]. Mir 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 könnten.

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, zum Beispiel bei der Registervergabe. Ein reeller Computer nur eine begrenzte Anzahl an Registern, somit können nicht unendlich viele Variablen gleichzeitig zur Benutzung 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 gebraucht werden. (Frei nach dem Drachenbuch[Aho08] 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 der 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. Drei Address Code:
   Diese Klasse soll alle Instruktionen sammeln um für sie Grundblöcke und Datenflussmengen zu berechnen.
4. Grundblöcke:
   Die Grundblockklasse soll speichern wo im Drei Address Code Programm einzelne Grundblöcke anfangen, aufhören und zu welchen Addressen sie springen.

Daraus ergibt sich folgendes Klassendiagramm(Abb. 2):

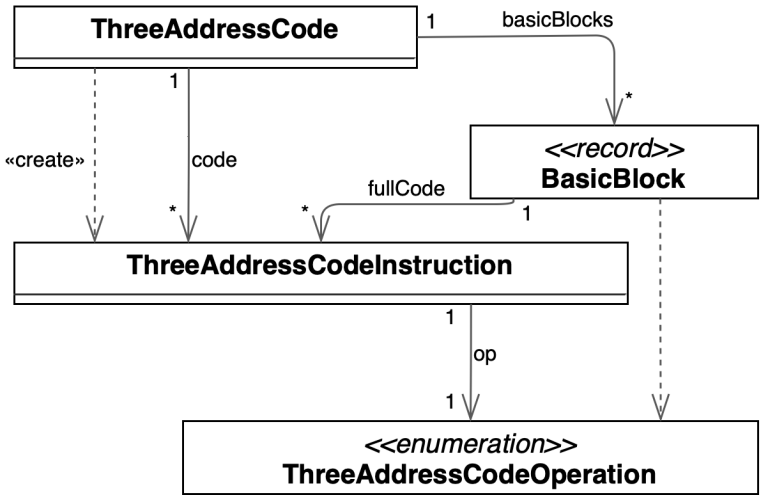

Abbildung 2: Die Drei Address Code Klassen im Überblick

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 2-Dimensionales 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.

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

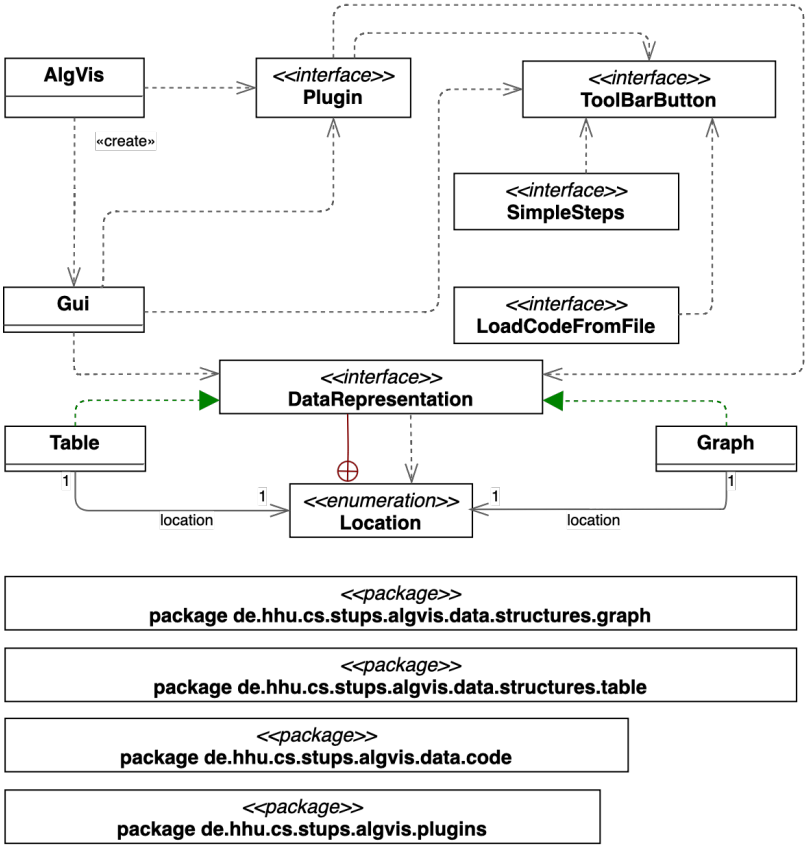

Abbildung 3: Klassenstruktur der GUI-Klassen

[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. Die verfügbaren grafischen Elemente sind Implementationen des Interfaces *DataRepresentation*, also entweder *Table* oder *Graph* auf die an einem späteren Zeitpunkt genauer eingegangen wird. 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 jedoch einen SplashScreen mit dem Schriftzug „welcome“an.
3. Und eine *JToolBar*.
   Da die meißten Plugins ähnliche Funktionalitäten haben, wie zum Beispiel das schrittweise Durchlaufen eines Algorithmuses, 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

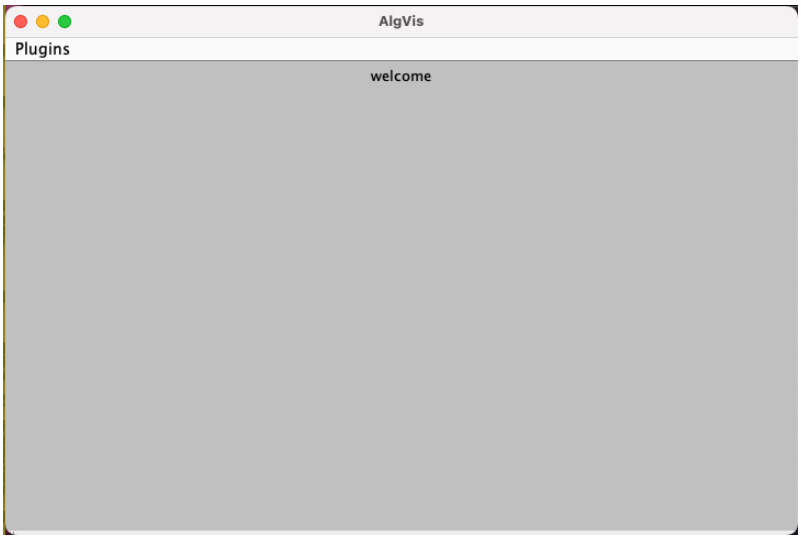

Abbildung 4: Ansicht des Programmes direkt nach Start

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

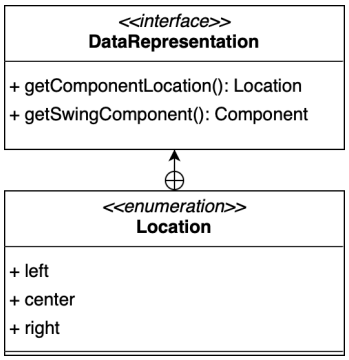

Abbildung 5: Das DataRepresentation Interface und sein Location enum

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

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.

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 4) 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

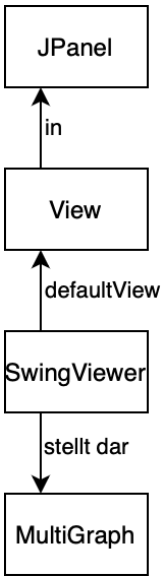

Abbildung 6: Rendering Pipeline eines Graphen

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(Srting, 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:

- 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 üi.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.

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

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

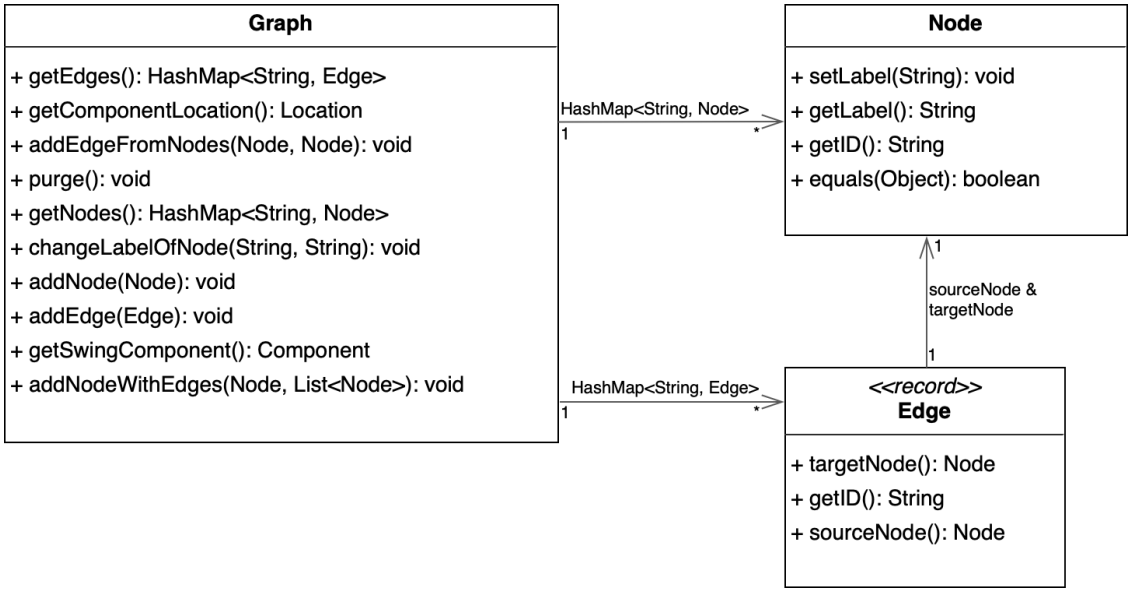

Abbildung 7: Klassenstruktur der Graph-Klassen

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 folgende Methode:

Listing 3: Generieren der ID für eine Kante

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

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. Wir erstellen also einen neuen Graphen wie folgt:

Listing 4: 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);
}
```

Die im Graph enthaltenen Knoten und Kanten werden in einem *Set* gespeichert, wenn ein Plugin dem Graphen einen Knoten hinzufügen möchte, ruft er die Methode *addNode(Node)*(Listing 5) 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.

Listing 5: Implementierung der addNode(Node) Methode

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

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 6)hinzugefügt.

Listing 6: Implementierung der setLabelOfNode(Node, String) Methode

```
public void setLabelOfNode(Node Node, String label){
  if(nodes.contains(newNode))
    graph.getNode(node).setAttribute("ui.label", label);
}
```

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.

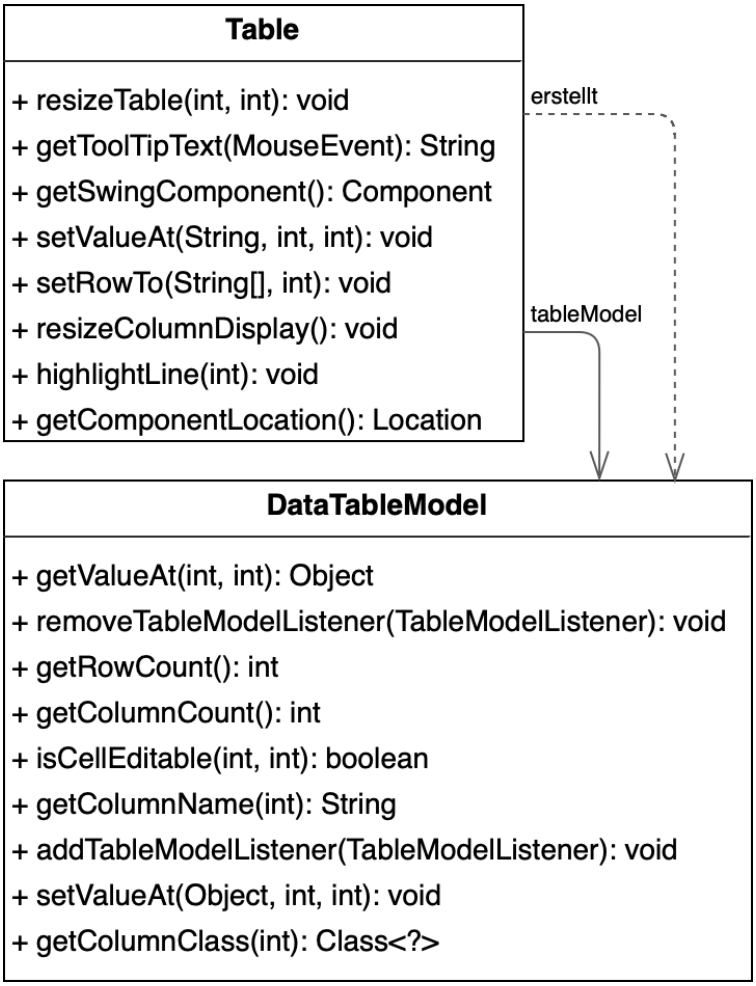

Abbildung 8: Klassenstruktur der Tabellenvisualisierung

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.

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.
- Die Methoden *addTableModeListener(TableModelListener)* und *removeTableModeListener(TableModelListener)* fügen eine Implementation der Klasse TableModelListener einer zugehörigen liste hinzu, beziehungsweise entfernen sie.
- *setValueAt(Object, int, int)* versucht den Wert am übergebenen Index zu überschreiben. Listing 7 Wenn dies möglich ist werden alle Objekte in der TableModelListener Liste benachrichtigt.

Listing 7: 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
  }
}
```

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

Die für Plugins verfügbar gestellten Methoden sind:

- *resizeTable(int, int)*
- *setValueAt(String, int, int)*
- *setRowTo(String[], int)*
- *highlightLine(int)*

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

Die Methode *getToolTipText(MouseEvent)*(siehe Listing 8) 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.

Listing 8: 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;
}
```

Die Methode *resizeColumnDisplay()*(siehe Listing 9) 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.

Listing 9: 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.getCo
```

```
        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
    }
```

2.3 Drei Address Code

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

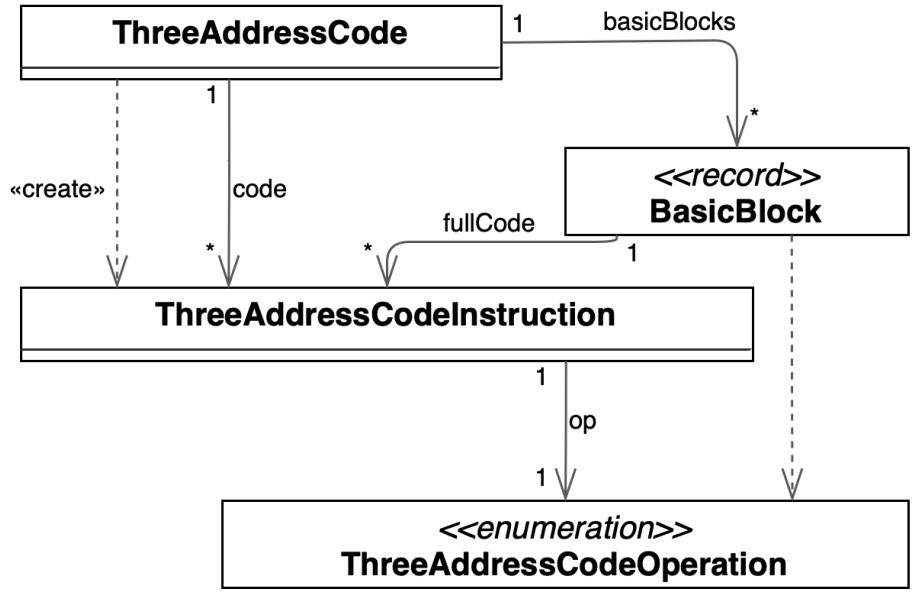

Abbildung 9: Implementation der drei Address Code Klassen

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 anzule-

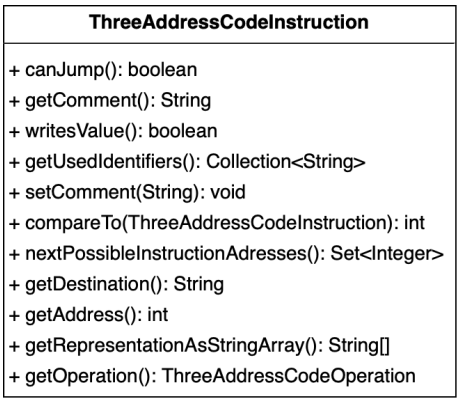

Abbildung 10: Die ThreeAddressCodeInstruction Klasse

gen.

Um die in Kapitel 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 zum hinzufügen von Kommentaren hinzugefügt.

Um aus dem Eingabestring ein passendes *ThreeAddressCodeInstruction*-Objekt zu generieren wird der Eingabestring im Konstruktor(Listing 10) der Klasse nach Leerzeichen aufgeteilt und wie folgt die richtige Operation ausgewählt:

Listing 10: 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
      }
    }
    ...
}
```

An den durch "..."markierten Stellen die zu gehörigen Attribute initialisiert.

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:

Listing 11: "Rückgabewert von getRepresentationAsArray(true)"

```
new String[]{address, destination, "=", source, op.getRepresentation()
```

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.

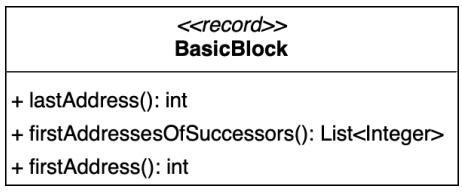

Abbildung 11: Die ThreeAddressCodeInstruction Klasse

2.3.3 Die ThreeAddressCode Klasse

2.4 Implementierung von Algorithmen

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

Hierbei gibt es zwei Interfaces die Implementiert werden, 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.

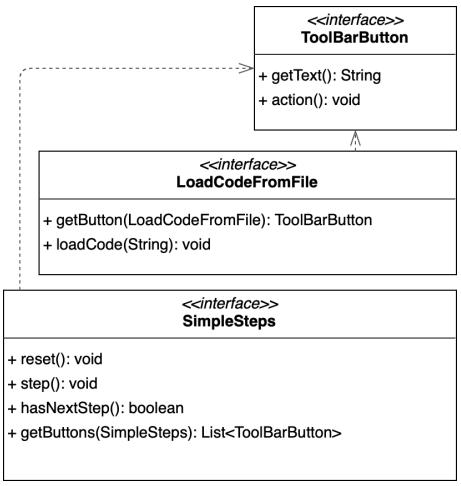

Abbildung 12: Das Plugin Interface

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 $JFileChooser$ [6] 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 implementiert 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.

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

2.4.1 Das Plugin Interface

Das $Plugin$ definiert die Hauptklasse des hinzugefügten Plugins.

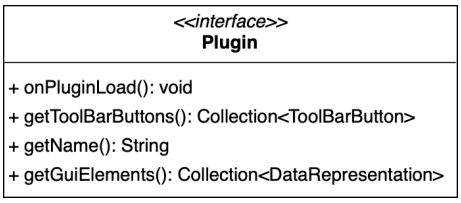

Abbildung 13: Das Plugin Interface

1. $onPluginLoad()$ 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.
2. $getToolBarButtons()$ gibt eine Collection der implementierten $ToolBarButton$s zurück, sodass diese wenn das Plugin geladen wird in die ToolBar eingefügt werden.
3. $getName()$ gibt den Namen des Plugins zurück, sodass er im Plugin Dropddown Menü angezeigt wird.
4. $getGuiElements()$ 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

Algorithmen wurden wie folgt implementiert:

3.1 Erstellung von Grundblöcken aus 3-Address-Code

3.2 Erstellung eines Kontrollflussgraphen aus 3-Address-Code

3.3 Analyse von erreichenden Definitionen

3.4 Analyse von lebendigen Variablen bezüglich Grundblöcken

3.5 Analyse von lebendigen Variablen bezüglich einzelnen Instruktionen

3.6 Optimierung von Grundblöcken mit Konstantenfaltung

konnte leider nicht implementiert werden.

4 Related Work

Alles

5 Evaluation

Alles

6 Future Work

Alles

7 Fazit

Alles

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

[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