



Compilerbau C--

Höhere Abteilung für Informationstechnologie

Ausgeführt von:

Stefan Kempinger, 5AHIT

Betreut durch:

Prof. OstR. Dipl.-Ing. Mag.
Franz Reitinger

In Zusammenarbeit mit der Johannes Kepler
Universität Linz

Betreuer: o.Univ.-Prof. Dipl.-Ing. Dr. Dr.h.c.
Hanspeter Mössenböck

Abgabe: 9. April 2019

Diplomand



Vorname: Stefan

Nachname: Kempinger

Geburtsdatum: 05.02.2000

Adresse: Adlhaming 39, 4655 Vorchdorf

E-Mail: mail@kempinger.xyz

Betreuer



Name: Prof. OstR. Dipl.-Ing. Mag. Franz Reitingner

Organisation: HTL Wels

Email: franz.reitingner@bildung.gv.at



Name: o.Univ.-Prof. Dipl.-Ing. Dr. Dr.h.c. Hanspeter Mössenböck

Organisation: JKU Linz

Email: hanspeter.moessenboeck@jku.at

Danksagung

Dank gebührt zuallererst der Johannes Kepler Universität in Linz, welche uns die Aufgabenstellung zur Verfügung gestellt hat. Herr o.Univ.-Prof. Dipl.-Ing. Dr. Dr.h.c. Hanspeter Mössenböck ist uns als Betreuer mit Rat und Tat immer zur Verfügung gestanden

Ich darf auch unserem Abteilungsvorstand, Herrn DI Stefan Svoboda, danken. Er hat mich über die Absolvierung eines Praktikums mit der Gelegenheit der Verfassung einer Diplomarbeit an der JKU umfassend informiert

Bedanken möchte ich mich schließlich noch bei meinem Betreuer an der HTL Wels, Herrn Prof. Mag. DI. Reitingner. Er hat mich bei den auftauchenden Problemen stets unterstützt. Weiters hat er die Vorgabe der JKU um die Durchführbarkeitsstudie für den Einbau von Elementen aus dem funktionalen Programmierparadigma erweitert.

Umstände des Projektes

Das Projekt wurde ursprünglich von zwei Diplomanden, Jakub Krajnansky und Stefan Kempinger, begonnen. Nach gesundheitlichen Problemen hat Jakub Krajnansky allerdings beschlossen, sich von dem Projekt zurückzuziehen. Er hat seinen Teil, den Interpreter, zum Zeitpunkt dieser Abgabe jedoch grundlegend abgeschlossen.

Eidesstattliche Erklärung

Hiermit versichern wir, die vorliegende Arbeit selbständig, ohne fremde Hilfe und ohne Benutzung anderer als der von uns angegebenen Quellen angefertigt zu haben. Alle Stellen, die wörtlich oder sinngemäß aus fremden Quellen direkt oder indirekt übernommen wurden, sind als solche gekennzeichnet.

Wels am _____ Name _____

Unterschrift _____

Abstract

C++ ist eine abgespeckte Version der weltweit am meisten verwendeten Programmiersprache C. An der JKU wird C++ für die Ausbildung (Compilerbau) eingesetzt und aus diesem Gebiet erhielten wir von der JKU (Institut für Systemsoftware) eine entsprechende Aufgabenstellung.

Zielsetzung

Im Rahmen der Diplomarbeit ist ein funktionstüchtiger Compiler für die Sprache C++ zu programmieren, der aus Textdateien eine Zwischendarstellung generiert. Diese wird wiederum mit Hilfe eines passenden Interpreters interpretiert und somit ablauffähiges Programm erzeugt. Diese Angabe wurde um die Erstellung einer graphischen Benutzeroberfläche zur Interaktion mit der Zwischendarstellung erweitert und durch die Erstellung eines Compilers für eine zweite Programmiersprache vollendet.

Geplantes Ergebnis

Geplant ist, dass die Implementierung des Compilers und der graphischen Benutzeroberfläche problemlos funktionieren. Diese sollen auf der Java Virtual Machine (JVM) ablauffähig und daher vollständig betriebssystemunabhängig ablauffähig sein.

Abstract (English)

C-- is a stripped-down version of the programming language C. A simple compiler for a simplified version of a known programming language is a common task in the computer science studies at universities, and thus this task was assigned to me.

Goal

This Project aims to develop a working Compiler for C-- which can generate an intermediate representation from text files that in turn can be turned into a running program by a fitting interpreter. This specification has been extended to create a graphical user interface to interact with the intermediate representation and finalized by creating a compiler for a second language.

Planned Result

It is planned that the implementation of both compilers and the graphical user interface will work completely. These should run on the Java Virtual Machine and be completely platform independent.

Zeitaufzeichnung

Zeitraum	Zeit in Stunden	Beschäftigung
Universitätspraktikum	8	Erklärung Projekt
	12	Grundlagen von Compilern (Recherche)
	4	Coco Einführung
	16	Schreiben einer eigenen Symboltabelle
	2	Revertieren zur gegebenen Symboltabelle
	18	Implementierung des Funktionstüchtigen C-- Compilers
	20	Debugging gesamt
Restliche Sommerferien	30	Scalaprogrammierung (Recherche)
	15	GUI
	3	JUnit Tests
	10	Ideenfindung zur 2. Sprache
Herbstsemester	26	Implementierung 2. Sprache incl. Funktionale Elemente
	41	Debugging und Verfeinerung gesamtes Projekt
	18	GUI Implementierung
	1	JUnit Tests
Frühjahrssemester	6	Implementierung der Kombination mehrerer Source Dateien
	37	Dokumentation

Inhaltsverzeichnis

1	AUFGABENSTELLUNG.....	10
2	AUFTEILUNG DES PROJEKTES	11
3	TECHNISCHES UMFELD	12
3.1	Low-Level oder Assemblersprachen.....	12
3.1.1	Einfluss der verwendeten Prozessorarchitektur	12
3.1.2	Arbeitsweise am Beispiel ADD	12
3.1.3	Notwendigkeit von Hochsprachen	13
3.2	Interpretierte Sprachen	13
3.2.1	Allgemeines Konzept	13
3.2.2	Konzept der intermediären Maschinen am Beispiel Java	13
3.3	Programmierparadigmen im Vergleich	14
3.3.1	Das Prozedurale Paradigma	14
3.3.2	Das Funktionale Paradigma.....	14
3.3.3	Das Objektorientierte Paradigma.....	14
4	TECHNOLOGIEN IM EINSATZ	15
4.1	Java [5]	15
4.2	Java FX [5]	15
4.3	Scala [6]	16
4.4	Junit [7]	16
4.5	Git [8]	16
5	COMPILER.....	17
5.1	Attributierte Grammatiken.....	17
5.1.1	Theoretische Grundlagen.....	17
5.1.2	Bedeutung im Compilerbau	18
5.1.3	Praktische Umsetzung.....	18
5.2	Coco/R.....	19
5.2.1	Theoretische Grundlagen [2]	19
5.2.2	Realisierung.....	19
5.3	Symboltabelle.....	20
5.3.1	Theoretische Grundlagen.....	20
5.3.2	Praktische Umsetzung.....	21

5.4	Syntaxbaum.....	23
5.4.1	Theoretische Grundlagen.....	23
5.4.2	Praktische Umsetzung.....	23
5.5	Scopes	24
5.5.1	Theoretische Übersicht	24
5.5.2	Praktische Umsetzung.....	24
5.6	Intermediate Representation (IR)	25
5.7	C-- Compiler.....	25
5.7.1	Übersicht.....	25
5.7.2	Implementierung.....	25
5.7.3	Probleme.....	25
5.8	Eine eigene Programmiersprache	26
5.8.1	Übersicht.....	26
5.8.2	Implementierung.....	26
5.8.3	Probleme.....	28
6	GRAPHISCHE BENUTZEROBERFLÄCHE.....	29
6.1	Übersicht	29
6.2	Implementierung.....	29
6.3	Probleme	32
7	ANHANG	33
7.1	Literaturverzeichnis	33
7.2	Abbildungsverzeichnis	33

1 Aufgabenstellung

Ziel dieses Projekts ist die Entwicklung eines Compilers und eines Interpreters für eine (leicht veränderte) Untermenge der Sprache C (genannt C--). Das Projekt soll im Rahmen von zwei Diplomarbeiten an der HTL Wels implementiert werden, wobei sich die Aufgaben wie folgt verteilen:

- **Compiler.** Mit Hilfe des Compilergenerators Coco/R (<http://ssw.jku.at/Coco>) soll ein Compiler entwickelt werden, der C-- Programme in eine Symbolliste und einen abstrakten Syntaxbaum (AST) übersetzt.
- **Interpreter.** Es soll ein Interpreter entwickelt werden, der abstrakte Syntaxbäume ausführt und als Laufzeitdatenstrukturen einen Prozeduren-Stack sowie einen Bereich für globale Daten verwaltet.

Die beiden Diplomarbeiten können parallel entwickelt werden. Die Schnittstelle zwischen Compiler und Interpreter bilden die Symbolliste und der abstrakte Syntaxbaum. Die Implementierungssprache des Projekts ist Java. Ein C-- Programm besteht aus einer einzigen Datei. Als Datentypen gibt es int, float, char (Ascii), string sowie (mehrdimensionale) Arrays und Structs. Als Anweisungen gibt es neben Zuweisungen und Prozeduraufrufen noch Verzweigungen und Schleifen. Prozeduren können void-Prozeduren oder Funktionen sein. Bei der Parameterübergabe wird Call by Value und Call by Reference unterstützt. Die Ein-/Ausgabe beschränkt sich auf das Lesen und Schreiben von Zeichen.

Das Projekt wird vom Institut für Systemsoftware der Johannes Kepler Universität Linz (<http://ssw.jku.at>) mitbetreut. Die beiden Maturanten verbringen in der Anfangsphase zwei Wochen am Institut, um mit den Techniken des Compilerbaus und der Interpretation von Programmen vertraut zu werden. Die Hauptbetreuung und Beurteilung der Arbeiten erfolgt seitens der HTL Wels.

Dieses Dokument ist derzeit lediglich eine Projektskizze, die die wesentlichen Eckpunkte der Aufgabenstellung festlegt. Eine detaillierte Projektbeschreibung soll im Rahmen des Projekts als Dokumentation entstehen.

2 Aufteilung des Projektes

Bei der Projektierung wurde von Anfang an auf eine strikte Aufteilung der Tätigkeiten und der zur Verfügung gestellten Ressourcen (Dateien) geachtet. Damit wird einerseits die Leistungsbeurteilung und andererseits eine Fortschrittskontrolle des Projektes ermöglicht.

Die Dateien wurden per Java Pakete getrennt. Die Paketstruktur sieht folgendermaßen aus:

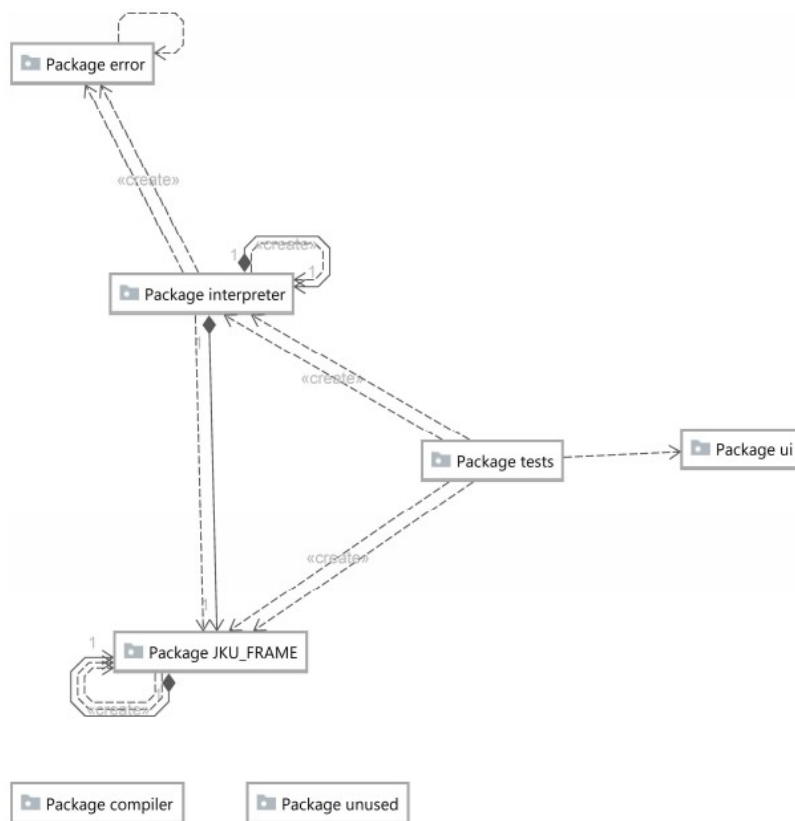


Abb. 1: Projektstruktur

3 Technisches Umfeld

Von komplexen, für Maschinen spezifisch geschriebenen Code bis hin zu Beschreibungen, welche Inhalte vom Gerät vollständig unabhängig darstellen: Programmiersprachen gibt es in den verschiedensten Ausführungen.

3.1 Low-Level oder Assemblersprachen

3.1.1 Einfluss der verwendeten Prozessorarchitektur

Bei den sogenannten Assemblersprachen ist ein wichtiger Punkt die Architektur von den darunterliegenden Prozessoren. Prozessoren die beispielsweise nach der x64 Architektur gebaut sind besitzen andere Instruktionen, auch OP-Codes genannt, wie Prozessoren die nach der x86 Architektur gebaut sind, und wiederum anders als die zumeist in mobilen Geräten verwendeten ARM Prozessoren.

Diese verschiedenen Instruktionen verkomplizieren das Implementieren von Programmen für verschiedene Plattformen, da man für das gleiche Ergebnis mehrere verschiedene Codes verwenden muss.

3.1.2 Arbeitsweise am Beispiel ADD

Let's start our introduction using a simple example. Imagine that we want to add the numbers from 1 to 10. We might do this in C as follows.

```
int total;
int i;

total = 0;
for (i = 10; i > 0; i--) {
    total += i;
}
```

The following translates this into the instructions supported by ARM's ISA.

```
MOV R0, #0      ; R0 accumulates total
MOV R1, #10     ; R1 counts from 10 down to 1
again ADD R0, R0, R1
SUBS R1, R1, #1
BNE again
halt B halt     ; infinite loop to stop computation
```

Abb. 2: ARM Assembly

Wie man erkennt, wird durch einen Compiler der logische C Code in systemabhängigen Assemblercode verwandelt.

3.1.3 Notwendigkeit von Hochsprachen

Diese Abhängigkeit wurde schnell als problematisch erkannt, und deshalb wurden Abhilfen geschaffen. Diese Abhilfen sind im allgemeinen als Hochsprachen bekannt. Solche Hochsprachen haben das grundlegende Ziel den Programmieraufwand an Programmen teils drastisch zu senken. Da diese Vereinfachung auch mit einer Generalisierung und somit teils überflüssigen Code einhergeht werden heute noch immer Teile wichtiger Programme plattformabhängig geschrieben. Viele der modernen Sprachen verwenden Compiler, um von einer Hochsprache, beispielsweise "C", in Assemblersprachen zu übersetzen.

3.2 Interpretierte Sprachen

3.2.1 Allgemeines Konzept

Interpretierte Sprachen zeichnen sich durch ein spezielles Programm aus, welches sie in einen drastischen Gegensatz zu Assemblersprachen oder auf Assemblersprachen kompilierte Programme stellt. Die Rede ist von einem Interpreter, welcher für verschiedene Plattformen implementiert ist, und auf diesen verschiedenen Plattformen eine gleichartige Umgebung zur Ausführung von Programmen erzeugt.

Eine Form dieser Umgebung ist eine virtuelle Maschine. Dieses Programm verwendet meist eine Sprache, die Assembler sehr ähnelt, aber einige Abstrahierungen enthält, um eine uniforme, plattformunabhängige Laufzeit für Programme zu schaffen.

3.2.2 Konzept der intermediären Maschinen am Beispiel Java

Eine der bekanntesten dieser virtuellen Maschinen ist die von der Oracle Corporation spezifizierte und auch implementierte Java Virtual Machine, kurz JVM. Diese ist durch automatisierter Optimierung teils performanter als vergleichbarer, nicht optimierter Code bekannter performanter Hochsprachen wie "C", was sie sehr attraktiv für große Programme macht. In der Sprache Java wird jedes Programm mithilfe des Java Compilers in JVM Bytecode, vergleichbar mit Assembler Code, übersetzt, um lauffähig zu sein. Ein Vorteil dieser Darstellung ist der Fakt dass der erzeugte Code auf jeder kompatiblen JVM gleich läuft und somit ein sehr großer Entwicklungsaufwand wegfällt.

3.3 Programmierparadigmen im Vergleich

3.3.1 Das Prozedurale Paradigma

Im Prozeduralen Paradigma basiert die Herangehensweise auf die Aufteilung der zu lösenden Probleme auf einzelne Blöcke/Funktionen und durch eine starke Trennung zwischen Funktionalitäten und Daten. Diese Art von Programmiersprachen ist auch teils synonym für die Imperative Programmierung, was den "klassischen" Aufbau und Ablauf beschreibt.

Programme, die in dieser Art verfasst wurden, eignen sich besonders für die Lösung von systemnahen Problemen.

Weiters zeichnet sie die Tatsache aus, dass sie die Schritte, die der Computer zum Erfüllen einer Anweisung benötigt, genau beschreibt. Daher spricht man auch vom imperativen Stil.

3.3.2 Das Funktionale Paradigma

Die am weitesten verbreiteten Ideen zur funktionalen Programmierung beziehen sich auf den Grundsatz, dass Programme nur Funktionen sind, welche aus einem gewissen gegebenen Input einen Output liefern. Funktionen sind dadurch gekennzeichnet, dass diese bei der selben Eingabe immer das gleiche Ergebnis liefern. Funktionen haben keine Seiteneffekte, d.h. die Funktionen speichern sich keine Zwischenstände.

Des Weiteren sind in der funktionalen Programmierung alle Teile von Programmen als Funktion zu betrachten, in einigen Implementierungen selbst konstante Werte.

Diese Ansicht steht als Gegensatz zur prozeduralen Programmierung, wo man dem Computer direkte und lineare Anweisungen gibt wie ein Programm durchzulaufen ist.

3.3.3 Das Objektorientierte Paradigma

Die Objektorientierung wurde lange Zeit als die Weiterentwicklung und die Ablöse der klassischen prozeduralen Programmierung betrachtet. Bei der Objektorientierung hat man einen eher organischen Ansatz der Behandlung von Daten. Diese werden in der Objektorientierung mit Methoden verknüpft, welche mit den Objekten interagieren und diese Daten verändern können. Diese Abstraktion, welche meist erst ausprogrammiert werden muss, führt zu einer großen Menge an teils redundanten Code, bekannt beispielsweise aus Java, wo man auf Daten nur mit dazu implementierten Methoden ändern sollte.

4 Technologien im Einsatz

4.1 Java [5]

Grundsätzlich wurde die Diplomarbeit vollständig für die Java VM realisiert, sodass das dahinter liegende Betriebssystem die notwendige JVM nur in der Version 8 unterstützen muss. Getestet wurde diese Betriebssystem-Unabhängigkeit bei jeder Ausführung des Programms, da die Diplomanden sowohl Linux als auch Windows verwendeten.

Obwohl ein sehr großer Teil in der Programmiersprache Java geschrieben ist, ähneln viele Stellen des Codes einer möglichen Implementierung in anderen Programmiersprachen. Meistens wurden nur Verweise auf andere "Datenklassen" verwendet. Diese Ähnlichkeit ist nicht zufällig, da COCO/R auch Compiler in anderen Programmiersprachen erzeugen kann und diese Art von Verweisen auf Datenstrukturen in vielen Programmiersprachen implementiert ist.

Ein weiterer Punkt, der bei der Auswahl der Sprache wichtig ist die Verwendung der eingebauten Persistierung, was in Java durch das Serialisieren von Objekten sehr einfach ist. Um eine solche Serialisierung zu ermöglichen muss man lediglich die zu serialisierenden Klassen von dem Interface "Serializeable" erben lassen, was der Java VM signalisiert dass jene Klasse serialisiert werden kann.

4.2 Java FX [5]

Die graphische Benutzeroberfläche verwendet als Framework "JavaFX", ein in dem offiziellen Java Development Kit enthaltenes GUI Framework. JavaFX ermöglicht den Benutzern die Realisierung von einfachen und modernen grafischen Bedienoberflächen.

JavaFX ist der moderne Ersatz von Swing und AWT, welcher von Oracle spezifiziert wurde. JavaFX ist vollständig unabhängig von diesen in die Jahre gekommenen GUI Toolkits.

Ein sehr wichtiges und in dieser Diplomarbeit auch verwendetes Feature ist die Möglichkeit Fenster sowohl programmatikalisch als auch mittels einer Seitenbeschreibungssprache, einer Version von XML, zu erstellen.

4.3 Scala [6]

Zusätzlich wurde bei dem GUI die Programmiersprache Scala verwendet. Scala ist ein komplexes aber äußerst vielseitiges Werkzeug, mit welchem sich in kurzer Zeit große Fortschritte erzielen lassen. Scala wurde allerdings wegen seiner schwierigen Lesbarkeit und der langen Compile-Zeiten nicht für das komplette Programm verwendet.

4.4 Junit [7]

Von beiden Diplomanden wurde während der Entwicklung das Framework JUnit in der Version 5.2 zum Testen eingesetzt, um neue Funktionalitäten zu testen oder um sicherzustellen, dass alles richtig abläuft.

JUnit war speziell zu Anfang des Projektes von entscheidender Wichtigkeit, da noch kein Interpreter zum Testen des Kompilierten Codes vorhanden war. Somit stellten Funktionen des Frameworks einen Ersatz für den Interpreter dar.

4.5 Git [8]

Zur Versionierung und zur einfachen Kollaboration wurde das Programm "git" in Verbindung mit der Plattform Github verwendet. Diese Kombination vereinfachte das Projekt stark, da Fehler im Code sehr schnell behoben werden konnten und beide Diplomanden immer am aktuellsten Stand waren.

5 Compiler

5.1 Attributierte Grammatiken

5.1.1 Theoretische Grundlagen

Attributierte Grammatiken werden verwendet, um die Grammatik(Syntax) von Programmiersprachen zu beschreiben. Anwendung findet diese Beschreibung meist bei Programmen, da die Korrektheit der zutreffenden Sprachbausteine programmgesteuert und formalistisch überprüft werden kann. Das aus der Grammatik abgeleitete Programm heißt Syntaxanalysator (Parser). Attributierte Grammatiken liefern also den Parser, um dann automatisiert zu erkennen, ob ein Programm (Eingabe) die zugrundeliegende Grammatik erfüllt. Daraus entsteht jedoch noch nicht ein neuer Compiler. Dafür braucht man noch semantische Aktionen und einen Compilergenerator welcher erst die vollständigen attributierten Grammatiken zu einem fertigen Compiler parst.

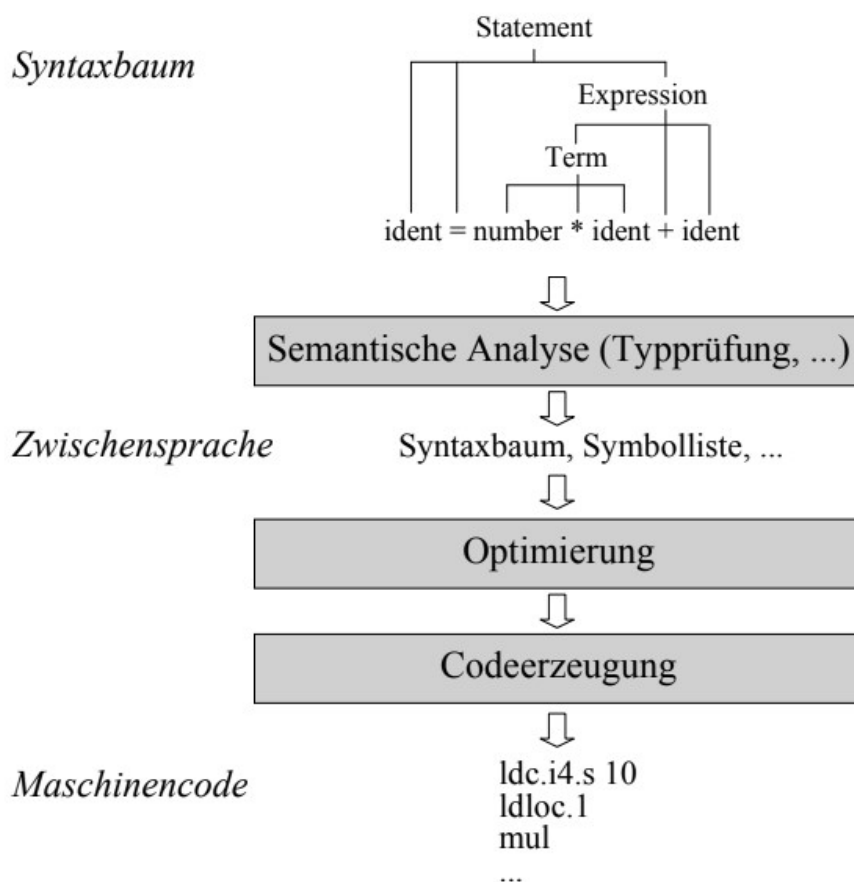


Abb. 3: Compilerbau [4]

5.1.2 Bedeutung im Compilerbau

Moderne Programmiersprachen werden zumeist am Anfang mit einer Attribuierten Grammatik dargestellt, um die gewünschten Features einer Sprache übersichtlich darstellen zu können.

5.1.3 Praktische Umsetzung

Im Zuge des Praktikums an der JKU wurde unserer Gruppe eine bereits fertige Datei zur Verfügung gestellt, welcher nur die Anweisungen zum Erstellen einer Symboltabelle mit Syntaxbäumen fehlten. Diese Anweisungen beinhalten Javacode welcher von Coco/R in die Parser.java Datei eingebaut wird.

```
VarDecl
=
Type<out Type type>
Ident<out String name>
(.symbolTable.insert(ObjKind.VAR,name,type);.)
{" " Ident<out String name1>
(.symbolTable.insert(ObjKind.VAR,name1,type);.)
}
";".
```

Dies ist die Beschreibung einer Variable. In der attribuierten Grammatik finden sich solche Beschreibungen für alle Elemente einer Programmiersprache.

Zu aller erst muss man den Namen des Elementes definieren, in diesem Fall "VarDecl". Diese besteht aus einem Typ und einem Identifier, also den Variablennamen. Falls eine solche Kombination in einem Programm entdeckt wird, wird die passende Variable in die Symboltabelle eingefügt. Falls nach dem Variablennamen ein Beistrich folgt wird noch weiter überprüft und beliebig viele weitere Variablen eingefügt. Zu guter Letzt wird noch zur syntaktischen Korrektheit das Vorhandensein eines Strichpunktes sichergestellt.

5.2 Coco/R

5.2.1 Theoretische Grundlagen [2]

Coco/R [CocoR] ist ein Werkzeug, das eine attributierte Grammatik in einen Scanner und einen Parser übersetzt, der nach der Methode des rekursiven Abstiegs [Wirth96] arbeitet. Der Benutzer muss noch ein Hauptprogramm hinzufügen, das den Parser aufruft, sowie Klassen, die von den semantischen Aktionen des Parsers benutzt werden (z.B. Klassen für die Symbolliste, die Codeerzeugung oder die Optimierung; siehe Abb. 1). Wir beschreiben hier die Version von Coco/R, die in C# implementiert ist und Scanner und Parser in C# erzeugt. Es gibt aber auch Versionen für Java, C/C++, Pascal, Modula und Oberon.

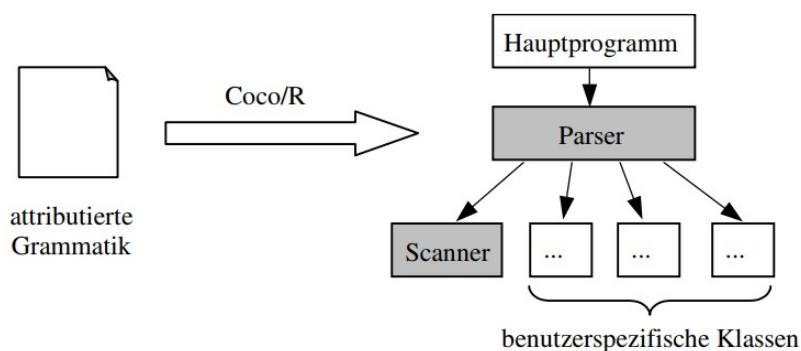


Abb. 4: Arbeitsweise von Coco/R [2]

5.2.2 Realisierung

Eine Vorgabe der JKU war auch die Verwendung des Compiler Generator Coco/R zu verwenden, welcher mithilfe spezieller attributierten Grammatik Dateien (.atg) einen Parser und einen Scanner generiert, welche verwendet werden um eine Quelltextdatei in eine Symboltabelle mit dazugehörigen Syntaxbäumen zu verwandeln.

Coco/R ist bereits in vielen Programmiersprachen implementiert und wir haben die für uns einfachste Variante genommen: die Java Implementierung.

Diese Version kann in der .atg Datei eingefügten Java Code parsen und diesen anschließend in einen Parser und Scanner in der jeweiligen Sprache verarbeiten.

Wir hatten mit Coco/R nach einer sehr kurzen Einführungsphase kaum Probleme, da dieses Produkt seit Jahrzehnten im Einsatz ist und somit vielfach getestet und verbessert worden ist.

5.3 Symboltabelle

5.3.1 Theoretische Grundlagen

Eine Symboltabelle wird von einem Compiler erstellt, um die einzelnen Teile eines Programmes, wie Variablen, Konstanten oder Prozeduren zwischenzuspeichern. Diese Symboltabelle wird im Falle unseres Projektes aufgebaut, während ein Scanner die Korrektheit des Programms überprüft.

Generell erfüllt eine Symboltabelle folgende Eigenschaften:

- Namen aller verwendeten Variablen speichern
- Sicherstellen, dass Variablen nicht doppelt deklariert werden
- Typen überprüfen
- Scopes verwalten

Eine Symboltabelle wird von den meisten Compilern erzeugt um eine Übersicht über die Symbole im Programm zu behalten.

Um einen möglichst effizienten Zugriff auf die Symbole zu ermöglichen wird in vielen Compilern die Elementlisten mittels Hashtabellen realisiert. Es ist in kleineren Programmen aber auch nicht verpönt einfache verlinkte Listen zu verwenden.

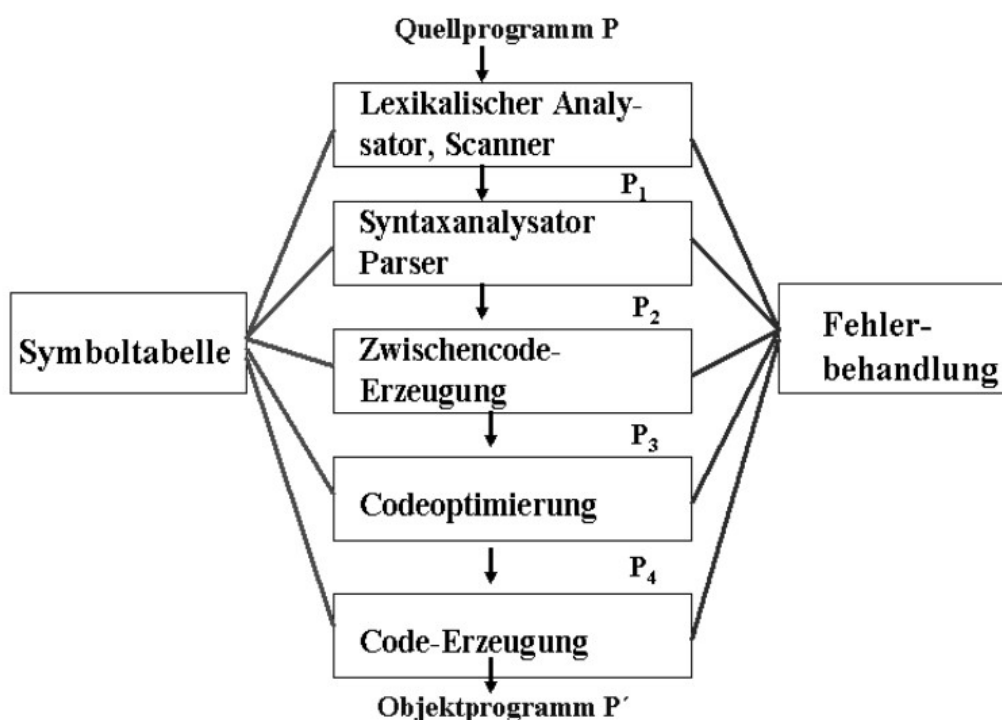


Abb. 5: Compilerüberblick [3]

5.3.2 Praktische Umsetzung

5.3.2.1 Übersicht

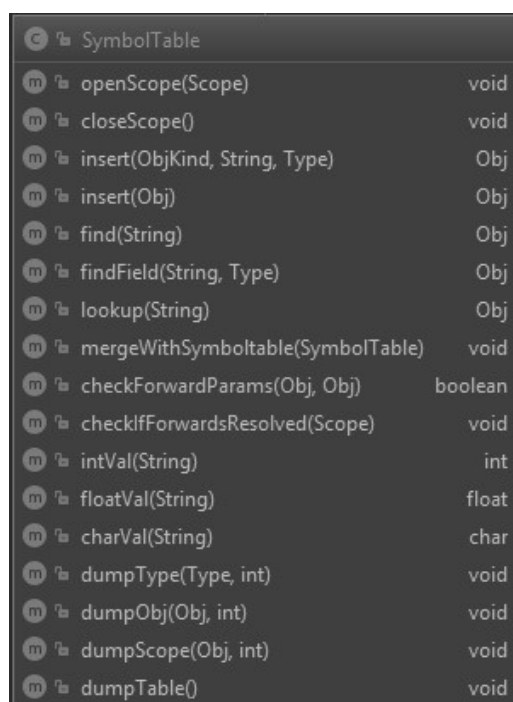
Ein Prototyp einer funktionstüchtigen Symboltabelle stand uns zur Verfügung. Dieser Prototyp wurde angepasst, um in der Realisierung der C-- Umgebung verwendet werden zu können. Größtenteils unverändert blieben lediglich die Methoden die zur Visualisierung der Daten verwendet werden. Diese Methoden wurden nur um die Möglichkeit der Verwendung mehrerer Scopes erweitert. Dies wurde im Original nicht berücksichtigt.

5.3.2.2 Funktionalität

Der Symboltabelle wird bei der Erstellung eine Instanz eines Parsers übergeben, welcher eine spezifische Datei verarbeitet.

Anschließend erzeugt sie den "universalen" Scope, welcher die vordefinierten Typen, wie Integer oder Float, beinhaltet und fügt diese ein. Diese vordefinierten Typen werden in weiterer Folge vom Programm verwendet.

Da die Symboltabelle für die strukturierte Speicherung von Programmteilen eingesetzt wird, sind die dementsprechenden Methoden zum Einfügen und Finden von Elementen implementiert.



SymbolTable	
openScope(Scope)	void
closeScope()	void
insert(ObjKind, String, Type)	Obj
insert(Obj)	Obj
find(String)	Obj
findField(String, Type)	Obj
lookup(String)	Obj
mergeWithSymboltable(SymbolTable)	void
checkForwardParams(Obj, Obj)	boolean
checkIfForwardsResolved(Scope)	void
intVal(String)	int
floatVal(String)	float
charVal(String)	char
dumpType(Type, int)	void
dumpObj(Obj, int)	void
dumpScope(Obj, int)	void
dumpTable()	void

Abb. 6: Symboltabellenmethoden

Das Einfügen erfolgt beim Durchlaufen des Parsers durch das zu kompilierende Programm. Es wird ein lookup durchgeführt, bei dem entweder ein gefundenes Objekt oder das vordefinierte "noObj", also kein Objekt, zurückgeliefert werden. Anschließend wird überprüft ob das Objekt vorwärts deklariert ist. Falls das zutrifft wird dem gefundenen Objekt (ein Prozedurkopf) der AST des einzufügenden Objektes hinzugefügt und das entstandene Objekt retourniert.

Falls das nicht zutrifft, wird überprüft ob kein "noObj" geliefert wurde. Dies bedeutet, dass dieses Objekt bereits existiert. Falls das zutrifft, wird eine Fehlermeldung ausgegeben.

Des weiteren wird überprüft ob das Objekt eine eigene Scope besitzt wie dies beispielsweise bei Schleifen zutrifft. Ein solcher Scope wird dann bei Zutreffen der Bedingungen hinzugefügt.

5.4 Syntaxbaum

5.4.1 Theoretische Grundlagen

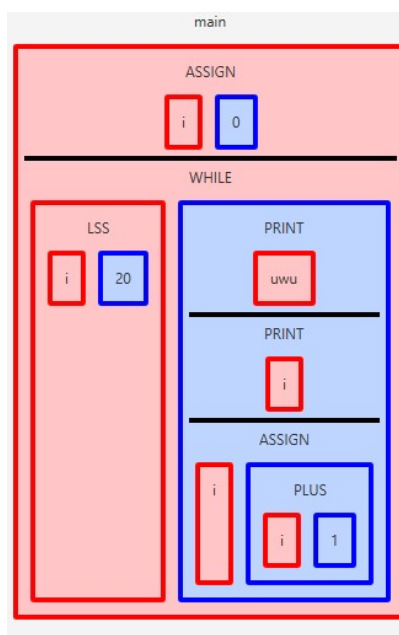
Für die Darstellung von Syntaxbäumen als Datenstruktur in Compilern wird weitestgehend die Bezeichnung abstrakter Syntaxbaum (engl: abstract syntax tree (AST)) recht einheitlich verwendet, wobei die Terminologie hier schwankt und z. B. ebenfalls von abstrakten Ableitungsbäumen, Operatorbäumen o. Ä. die Rede sein kann.

Syntaxbäume werden von Compilern erzeugt um die einzelnen Bestandteile von Prozeduren hierarchisch darzustellen. Diese Bestandteile sind beispielsweise IF-Verzweigung oder WHILE-Schleifen.

5.4.2 Praktische Umsetzung

In der Diplomarbeit werden Syntaxbäume durch die Klasse "Node" repräsentiert. Die Klasse enthält Verweise auf die verschiedenen Arten von Nodes, wie Assign oder auch Variablen und Verweise auf mögliche Nachfolger (Kind - Nodes) oder die weiteren Elemente der Verknüpfungen.

Dieser Syntaxbaum wird in der Symboltabelle mit dem Element PROC bezeichnet, sodass der Interpreter weiß, dass er auf den AST der Node zugreifen muss.



Dieses Beispiel zeigt den Syntaxbaum der folgenden Prozedur, geschrieben in "C":

```
void main() {
    int i;
    i=0;
    while(i<20){
        print("uwu");
        print(i);
        i=i+1;
    }
}
```

Abb. 7: Eine Symboltabelle

5.5 Scopes

5.5.1 Theoretische Übersicht

Mit Scope wird der Gültigkeitsbereich einer Variable bezeichnet. Die gängigsten Scopes, die in den meisten Programmiersprachen implementiert sind, sind der universale Bereich, für vordefinierte Werte, der globale Scope, in dem vom Programmierer Variablen definiert werden die im gesamten Programm gültig sind, und der funktions- oder prozedurspezifische Bereich, wo man von Prozeduren verwendete Variablen definiert, welche nur von innerhalb einer Prozedur benutzt werden können.

5.5.2 Praktische Umsetzung

In den meisten modernen Programmiersprachen sind allerdings Scopes meist mit diversen Programmblöcken eng verbunden, da die geschwungenen Klammern ("{" , "}") oftmals eine eigene Scope beschreiben.

Da C++ standardmäßig dieses Verhalten nicht unterstützt, sondern auf der "klassischen" Herangehensweise mit den drei Scope Levels beruht, wurde dieses Verhalten geändert.

Die im Rahmen dieser Arbeit realisierte Version von C++ hat einen Verweis auf die äußere Scope, einen Verweis auf die scopespezifischen Variablen und eine Variable, welche die Größe der enthaltenen Variablen beschreibt.

```
public class Scope implements Serializable {  
    public Scope outer;    // to outer scope  
    public Obj locals;    // to local variables of this scope  
    public int size = 0;   // total size of variables in this scope  
}
```


5.6 Intermediate Representation (IR)

Eine Intermediate Representation ist die Darstellung des Zwischenergebnisses welches ein Compiler erzeugt. Diese Darstellung unterscheidet sich je nach Compiler. Eine IR wird erzeugt um Optimierungen vorzunehmen oder wie in dem Fall dieser Diplomarbeit direkt interpretieren zu lassen.

Eine gute IR lässt sich durch die Tatsache erkennen, dass der Quellcode eine direkte Repräsentation besitzt und diese absolut akkurat, also fehlerfrei, ist.

Generell lässt sich sagen, dass kein guter Compiler vollständig ohne IR auskommt, sei es wegen der Übergaben von Daten an einen Interpreter oder bei der Optimierung von Programmen.

5.7 C-- Compiler

5.7.1 Übersicht

Für den C-- Compiler stand eine vorgegebene attributierte Grammatik zur Verfügung.

Für die Realisierung des C-- Compiler musste in der zur Verfügung gestellten attribuierten Grammatik spezielle Einträge für den Compiler Generator hinzugefügt werden, wobei dann mit Hilfe von Coco die notwendigen Scanner und Parser für die Sprache C-- erzeugt werden konnten.

5.7.2 Implementierung

Die Implementierung von dem Compiler für C-- war vergleichsweise schnell abgeschlossen, da von der JKU bereits eine Attributierte Grammatik und die gewünschte Symboltabelle zur Verfügung gestellt wurden. Zu Beachten ist, dass die Dateien teils stark verändert wurden, um für diese Diplomarbeit relevant zu sein.

5.7.3 Probleme

Dank der vorgegebenen Dateien war ein kompletter C-- Compiler sehr schnell und problemfrei implementiert.

5.8 Eine eigene Programmiersprache

5.8.1 Übersicht

Nach der funktionstüchtigen Implementierung der Sprache C-- war dieser Teil als abgeschlossen zu werten. Um also weitere Sprachfeatures hinzufügen zu können musste eine neue Sprache erstellt werden.

Diese neue, eigene Sprache wurde treffend als Language2 bezeichnet, da sie die zweite in der Diplomarbeit ausgearbeitete Sprache ist.

Die Sprache hat eine hohe lexikalische Ähnlichkeit mit C--, da die Sprache auf den grundlegenden Features von der bereits ausgearbeiteten Programmiersprache aufbaut.

5.8.2 Implementierung

Die Implementierung wurde an einer Kopie der C-- Dateien durchgeführt, wobei beachtet wurde dass möglichst kein Code doppelt vorkommen soll. Es wurden somit alle gemeinsamen Dateien in eigene Pakete ausgelagert und die beiden Sprachen auch in getrennte Pakete geschoben.

Es wurden während der andauernden Entwicklung der Sprache viele Features implementiert, getestet und wieder verworfen, was die Implementierung oft verzögerte.

Ein Beispiel für ein solches verworfenes Feature ist das Ersetzen der drei verschiedenen Klammerarten ("`[`", "`{`", "`(`") durch die einfachen ("`(`") Klammern. Dieses Feature war zwar funktionstüchtig, allerdings so unübersichtlich, dass wieder auf die klassische Representation umgestellt wurde.

Etwas, das sich als schwierig bis unmöglich mit COCO zu realisieren herausstellte war die Möglichkeit auf Strichpunkte ("`;`") am Ende von Statements zu verzichten. Diese Anfangs trivial erscheinende Hürde konnte (eigentlich) nicht genommen werden, da der Parser "`\n`", also das Zeichen für eine neue Zeile als Whitespace Zeichen herausfiltert. Eigentlich deshalb, da durch einen (mittlerweile) beabsichtigten Fehler in der Konstruktion der Symboltabelle auch Statements mit fehlendem Strichpunkt am Ende korrekt kompiliert werden.

Eine der wichtigsten Herausforderungen bei der Erfüllung der Aufgabenstellung für die zweite Sprache war die Implementierung von funktionalen Features. Durch die Möglichkeiten von Coco waren „foreach“ Schleifen und ein „filter“, schnell implementiert, welche Arrays durchlaufen

sollten. Man musste nur entsprechende Funktionen durch die bereits implementierten Funktionen wie Schleifen realisieren. Mit einer "while" Schleife und einer dynamischen Iteratorvariable stellt man in dieser Implementierung eine foreach Schleife dar. Zur Realisierung eines Filters verwendet man die gleiche Funktionalität, lediglich erweitert mit einer "if" Verzweigung.

Wichtig ist auch die Überlegung, ob ein Nullcheck vor der Ausführung des Inhalts der foreach Schleife durchgeführt werden soll, da bei vielen Interpreter implementierungen Felder von Arrays mit 0 vorangefüllt werden.

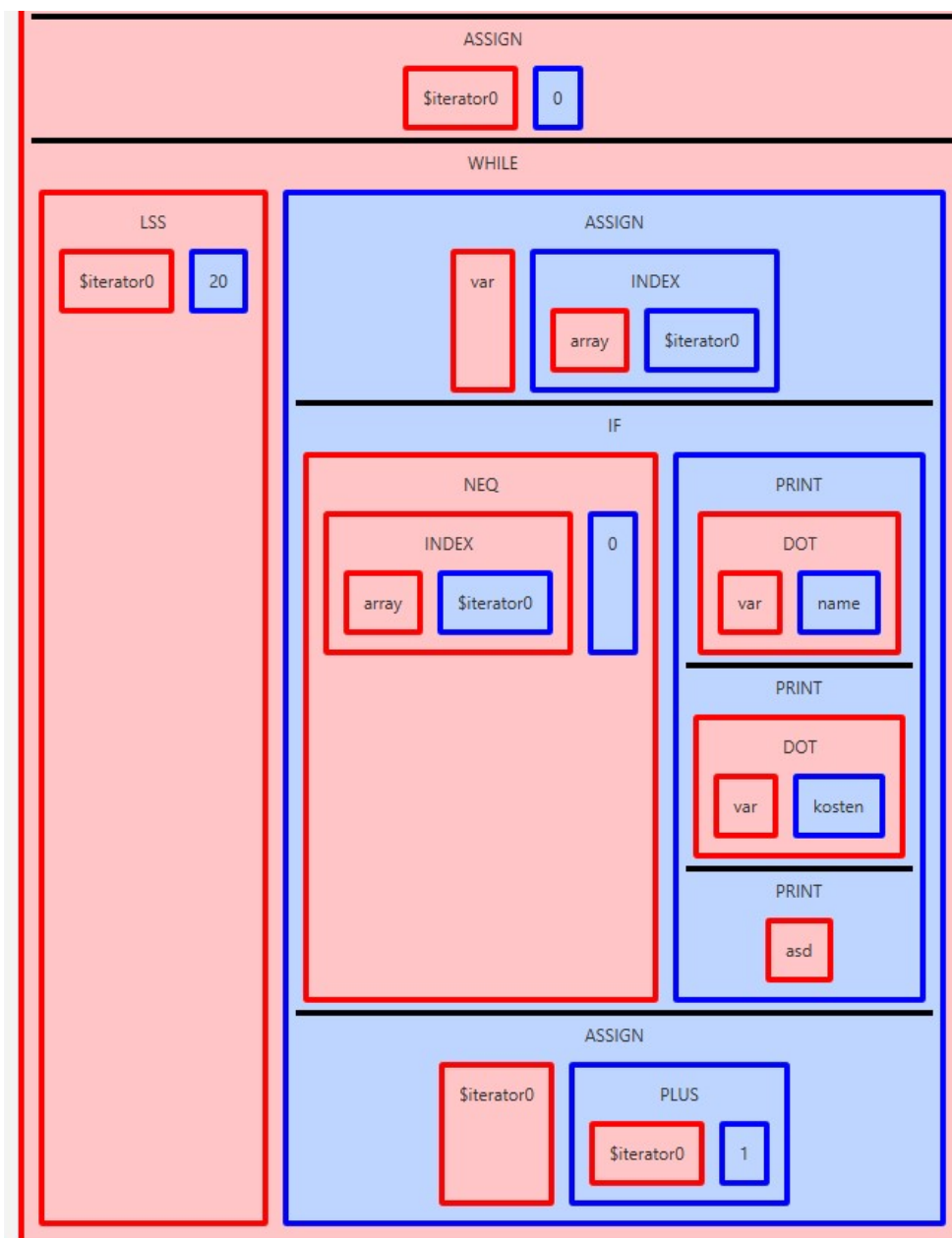


Abb. 8: Darstellung einer Foreach-Schleife

5.8.3 Probleme

Die Implementierung der eigene Programmiersprache stellte von Anfang an mehr Probleme dar, weil ich nicht mit einer bekannten Grammatik arbeitete, sondern diese erst während der Implementierung definieren musste.

Zusätzlich war es schwierig weil ich schnell an die Grenzen des Interpreters gelangte, womit sich spezielle Features wie Objektorientierung als sehr sehr schwierig zu implementieren herausstellten.

Nachteilig bei der Implementierung war die Implementierung des Interpreters, welcher jedes Feld mit 0 befüllt. Diese Befüllung vernichtet viele Möglichkeiten der Ausprogrammierung der funktionalen Elemente.

6 Graphische Benutzeroberfläche

6.1 Übersicht

Die grafische Benutzeroberfläche dient zur Visualisierung und dem Bearbeiten der erstellten Syntaxbäume. Diese stellen jeweils einzelne Prozeduren der aus der Quelltextdatei generierten Anwendung dar.

Man kann mit dieser grafischen Benutzeroberfläche einzelne Elemente wie Konstanten oder Operatoren verändern, man kann aber auch Elemente löschen.

Als Technologie wurden JavaFX und Scala verwendet um die Oberfläche schnell und effizient erstellen zu können. Zusätzlich ermöglicht Scala die Automatisierung von vielen Bestandteilen der JavaFX Applikation mit Hilfe von Makros, welche Einmalig ausprogrammiert vielfach verwendet werden können.

6.2 Implementierung

Die Makros von Scala wurden definiert und in eine eigene Datei (GUI_Generator.scala) ausgelagert.

Diese Makros werden an die JavaFX verwendenden Scala Klassen vererbt, sodass sie verwendbar sind.

```
trait GUI_Generator {  
  def generate  
  
  implicit class OnClickMakro(node: javafx.scene.Node) {  
    def onClick(function: => Unit) =  
      node.setOnMouseClicked(event => function)  
  }  
  
  implicit class GetChildrenMakro(pane: Pane) {  
    def addChild(child: javafx.scene.Node) =  
      pane.getChildren.add(child)  
  }  
}
```



Abb. 9: Menü der Benutzeroberfläche

Das Menü des Programms besitzt ein Feld zum Einfügen des Dateinamen, wobei dieses Feld von dem Ordner "testfiles/" im Projektordner ausgeht.

Neben diesem Feld befindet sich ein Button zum Parsen der Datei. Dieser Parsevorgang ist nur ein graphischer Zugriff auf die Funktion `Helpers.parseSymboltable()` in der `Helpers.scala` Datei. Dabei wird der Mechanismus zum Parsen durchgeführt.

Der Button mit dem Titel "Show AST" zeigt die Darstellung der vorher geparsen Symboltabelle. Der Button ist nicht Funktionstüchtig, wenn vorher keine Datei geparst wurde.

Des weiteren gibt es den Button "Interprete", welcher den Interpreter aufruft mit der Symboltabelle.

Da die graphische Benutzeroberfläche für Veränderungen an den Syntaxbäumen der einzelnen Prozeduren verwendet werden kann ist es wichtig, dass die Instanz der Symboltabelle im Programm gespeichert und von den einzelnen Funktionen nur referenziert ist.

Mit diesem Mechanismus kann man graphisch Veränderungen am Programm vornehmen, und diese gleich ohne Neukompilierung testen.

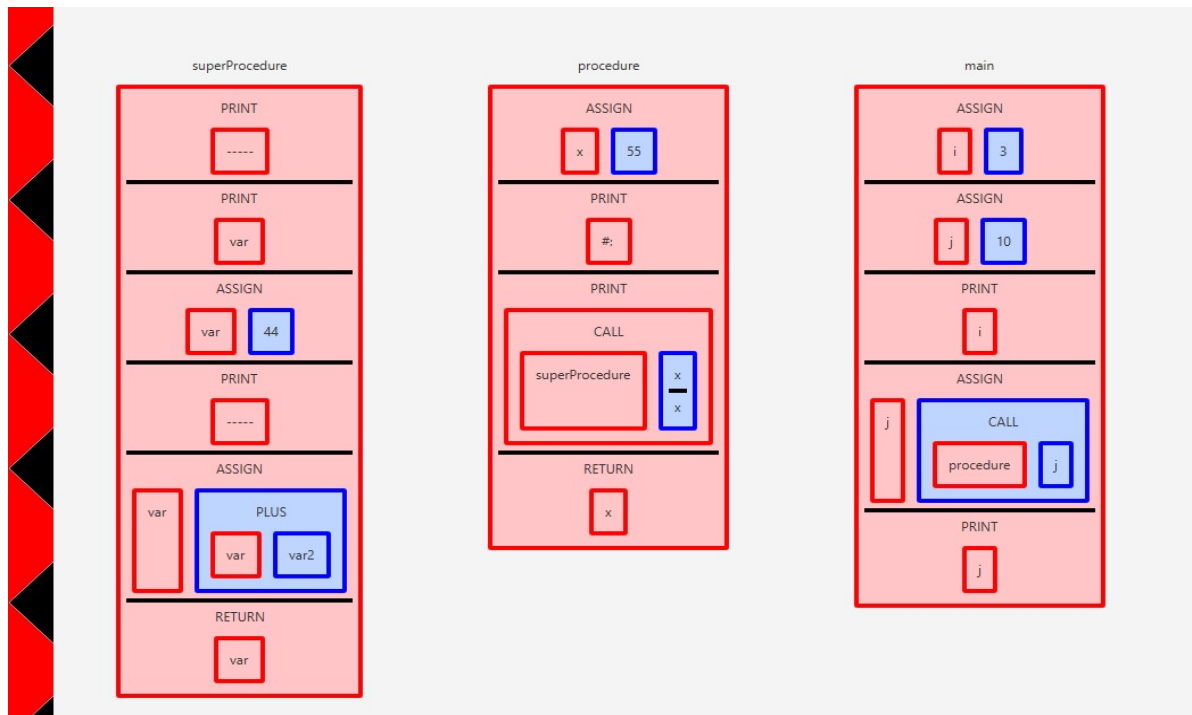


Abb. 10: Darstellung Syntaxbäume

Das Darstellen in der graphischen Benutzeroberfläche ermöglicht einen Überblick über das Programm und hilft Fehler frühzeitig zu erkennen und diese vor dem Ausführen noch zu beheben.

```
int superProcedure(int var, int var2){
    print("-----");
    print(var);
    var=44;
    print("-----");
    var=var+var2;
    return var;
}

int procedure(int var){
    int x;
    x=55;
    print("----# #");
    print(superProcedure(x x));
    print("----# #");
    return x;
}

void main(){
    int i;
    int j;
    i=3;
    j=10;
    print(i);
    j=procedure(j);
    print(j);
}
```

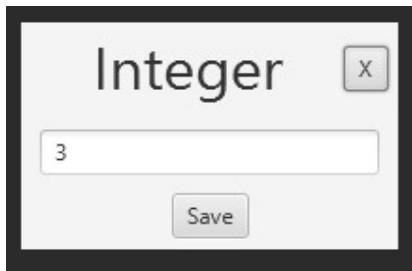


Abb. 11: Fenster zum Bearbeiten

Das Fenster zum Ändern von Konstanten ist implementiert, um jegliche Art von konstanten Werten, wie Integer, Float oder String, verändern zu können. Diese Änderung kann beispielsweise bei Schleifen oder anderen Kontrollstrukturen verwendet werden, um die Bedingungen zu anderen Ergebnissen zu zwingen.

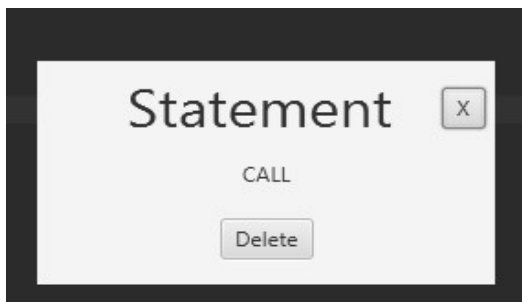


Abb. 12: Weiteres Fenster zum Bearbeiten

Einzelne Statements können in der graphischen Oberfläche herausgelöscht werden, um fehlerhafte Aufrufe vor der Ausführung noch herausnehmen zu können.

6.3 Probleme

Da die Implementierung mit Scala und JavaFX in Kombination eher außergewöhnlich ist, gibt es dazu eher wenig an Dokumentation und Forenbeiträge. Glücklicherweise beschränkten sich jegliche Probleme, wie das Fehlen einer Laufzeitbibliothek unter Linux, auf das bestehende JavaFX Framework.

7 Anhang

7.1 Literaturverzeichnis

- [1] Burch, Carl: Introducing ARM assembly language
<http://www.toves.org/books/arm/> [04.04.19]
- [2] Mössenböck, Hanspeter: Der Compilergenerator C—
<http://www.ssw.jku.at/Research/Papers/Moe03b/MoeWoeLoe03.pdf> [04.04.19]
- [3] http://www.saar.de/~awa/compiler_ueberblick.html
- [4] <http://ssw.jku.at/General/Staff/ML/Teaching/CC2007RO/downloads/CC2007RO.pdf>
- [5] <https://www.java.com/de/>
- [6] <https://scala-lang.org/>
- [7] <https://junit.org/junit5/>
- [8] <https://git-scm.com/>

7.2 Abbildungsverzeichnis

Abb. 1: Projektstruktur	11
Abb. 2: ARM Assembly	12
Abb. 3: Compilerbau [4]	17
Abb. 4: Arbeitsweise von Coco/R	19
Abb. 5: Compilerüberblick [3].....	20
Abb. 6: Symboltabellenmethoden.....	21
Abb. 7: Darstellung einer Foreach-Schleife	27
Abb. 8: Menü der Benutzeroberfläche.....	30
Abb. 9: Darstellung Syntaxbäume	31
Abb. 10: Fenster zum Bearbeiten.....	32
Abb. 11: Weiteres Fenster zum Bearbeiten	32



DIPLOMARBEIT

Compilerbau C--

Höhere Abteilung für
Informationstechnologie



Compilerbau C--