

Fachbereich Informatik
Fachhochschule Mannheim
Hochschule für Technik und Gestaltung

Diplomarbeit
im Studiengang Informatik

vorgelegt von
Harry Hartmut Dietz
2000

**Untersuchungen zum universellen
Register-Maschinen-Programm**

**Die Diplomarbeit wurde ausgeführt im
Fachbereich Informatik
unter der Betreuung von
Herrn Prof. Dr. H. Schnitzspan**

Ich erkläre, dass ich die Arbeit selbstständig und nur mit den angegebenen Hilfsmitteln angefertigt habe und dass alle Stellen, die dem Wortlaut oder dem Sinne nach anderen Werken entnommen sind, durch Angabe der Quellen als Entlehnungen kenntlich gemacht worden sind.

Inhaltsverzeichnis

I.	Einleitung.....	6
	Namenskonvention und wichtige Anmerkungen.....	9
II.	Die Register-Maschine	10
	1. Geschichte der Register-Maschine	10
	2. Definition der Register-Maschine.....	10
	3. Arbeitsweise der Register-Maschine	14
	4. Andere Definitionen der Register-Maschine	16
	5. Der Interpreter von Register-Maschinen-Programmen	20
	5.1. Die Vorbereitungen	20
	5.2. Der Interpreter	21
	5.3. Der Interpreter II.....	23
III.	Der Compiler	26
	1. Generelle Methodik	30
	2. Verwendete C++ Syntax.....	30
	3. Übersetzung der C++ Syntax.....	33
	3.1. Variablen und Funktionen	33
	3.2. Übersetzung von Zuweisungen und Rechenoperationen.....	35
	3.2.1 Übersetzung von „++“ und „--“.....	35
	3.2.2 Übersetzung von einfachen Zuweisungen	35
	3.2.3 Übersetzung von modifizierenden einfachen Zuweisungen.....	35
	3.2.4 Übersetzung von Rechenoperationen	38
	3.3 Übersetzung von „if“ / „while“ und Bedingungen	38
	3.3.1 Übersetzung von Bedingungen.....	38
	3.3.2 Übersetzung von „if“	40
	3.3.3 Übersetzung von „while“	40
	4. Übersetzer – Ausblick.....	42
IV.	Das universelle Register-Maschinen-Programm	44
V.	Zusammenfassung und Ausblick.....	50
VI.	Literatur- & Quellenverzeichnis	51
VII.	Verzeichnis der Abbildungen und Tabellen	52
VIII.	Glossar	53
IX.	Danksagungen.....	55
X.	Anhang.....	56
	1. Klasse CRegister.....	56
	1.1. Vollständige Übersichtstabelle der Klasse	56
	1.2. CRegister.cpp.....	57
	1.3. CRegister.h	67
	2. Laufzeitoptimierter Interpreter für Register-Maschinen-Programme	68
	2.1. D05.cpp.....	68
	2.2. Interpret.cpp.....	70
	2.3. Interpret.h.....	74
	2.4. Convert.h	74
	3. Interpreter für Register-Maschinen-Programme.....	75
	3.1. D03.cpp.....	75
	3.2. Convert.cpp.....	76
	3.3. Convert.h	79
	3.4. Interpret.cpp.....	80
	3.5. Interpret.h.....	83
	4. Compiler	83

4.1.	Vollständige Übersicht der Klassen und Strukturen.....	83
4.2.	D04.cpp.....	88
4.3.	CParser.cpp	88
4.4.	CParser.h.....	106
4.5.	CToken.cpp.....	109
4.6.	CToken.h	113

There is an infinite set a that is not too big.
John von Neumann (1903-1957)

I. Einleitung

Register-Maschinen werden in keiner Firma und in keinem Haushalt verwendet. Sie sind ein Gedankenmodell der Mathematik und Informatik. Register-Maschinen sind nahe Verwandte der wesentlich bekannteren Turingmaschinen, deren Vorgeschichte etwa um 1900 begann.

Der deutsche Mathematiker David Hilbert (1862-1943) schlug 1900 auf dem internationalen Mathematiker-Kongress in Paris vor, die Mathematik auf eine unangreifbare Grundlage von einigen wenigen Axiomen zu stellen. Diese Axiome sollten voneinander unabhängig und nicht mehr zu vereinfachen sein. Ausgehend von ihnen sollten alle mathematischen Sätze nacheinander bewiesen werden können. Er formulierte die Hypothese, dass es ein irgendwie geartetes Verfahren geben könnte, das jeden mathematischen Satz formuliert und automatisch beweist. Diese These ist unter dem Namen „Entscheidungsproblem“ in die Forschung eingegangen.

1931 aber fand der österreichische Mathematiker Kurt Gödel (1906-1978) den Unvollständigkeitssatz. Dieser besagt, dass es Systeme gibt, in denen mathematische Sätze existieren, die wahr sind, aber deren Wahrheit in diesem System nicht bewiesen werden kann. Gödel bewies dies insbesondere für die Peano-Axiome. Damit kann das Entscheidungsproblem also für unsere gewöhnliche Mathematik nie vollständig gelöst werden.

Um das Entscheidungsproblem näher untersuchen zu können, betrachtete der britische Mathematiker Alan Mathison Turing (1912-1954) um 1936 die Eigenschaften von allgemeinen Verfahren oder Algorithmen. Die Rechenmaschinen hatten damals schon eine etwa 300 jährige Geschichte hinter sich, von Blaise Pascals (1623-1662) Rechenmaschine, die subtrahieren konnte, über Charles Babbages (1782-1871) „Analytical Engine“ zu Hermann Holleriths (1860-1929) Apparat zur Auswertung der Volkszählung. Alle diese Rechenmaschinen arbeiteten allerdings mit festen Algorithmen, um Rechnungen auszuführen. Die Weiterentwicklung der „Analytical Engine“ von Charles Babbage hatte aber schon Ansätze der Programmierung, ebenso wie die Webmaschinen von Joseph Marie Jacquard, die durch Lochkarten gesteuert wurden.

Turing erdachte eine theoretische Maschine, die später seinen Namen erhielt: die Turingmaschine. Sie kann eine irgendwie geartete Eingabe in eine Ausgabe umwandeln, wobei Eingabe und Ausgabe aus einer Kette von Symbolen bestehen, die auf einem Band abgelegt sind. Die Maschine kennt keine arithmetischen Operationen, sondern innere Zustände, eine Tabelle mit Übergangsregeln von einem Zustand zu einem anderen und einen sogenannten „Schreib-Lese-Kopf“, der sich auf dem Band

bewegen kann und von diesem liest und darauf schreibt. Die Turingmaschine ist also eine programmierbare Maschine, wenn auch nur eine theoretische abstrakte.

Man kann sich diese Maschine etwa wie ein Tonbandgerät vorstellen. Dabei enthält das Magnetband, von dem gelesen wird, die Eingabe. Gleichzeitig ist dieses Magnetband, auf das dieses Tonbandgerät aufzeichnen oder schreiben kann auch die Ausgabe. Die Arbeitsweise ist nun folgende: Die Maschine liest ein Symbol der Eingabe, die Übergangsregeln sagen ihr in welchen Zustand sie wechseln soll, welches Symbol auf die Ausgabe geschrieben wird und in welche Richtung sich der Schreib-Lese-Kopf auf dem Band bewegen soll.

Um einen Algorithmus für eine Turingmaschine zu erstellen, muss also nur eine Tabelle mit Übergangsregeln aufgestellt werden. Solch eine Übergangsregel lautet dann etwa: „wenn die Maschine im Zustand A ist und das Zeichen X liest, dann wechsele in Zustand B, schreibe Zeichen Y und bewege den Schreib-Lese-Kopf nach links“. Solch eine Tabelle kann man auch als Turingmaschinenprogramm bezeichnen. Dabei ist das Programm, die Tabelle mit den Übergangsregeln, für die ganze Laufzeit der Maschine fest. Die Turingmaschine muss „von Hand“ gestartet werden, sie hält allerdings an, wenn ein Übergang zum Zustand „STOP“ erreicht wird.

Bei der Untersuchung verschiedener Algorithmen für Turingmaschinen, stellt sich die Frage, ob ein Algorithmus (ein Programm) in endlicher Zeit ein Ergebnis liefert oder nicht. Dieses sogenannte Halteproblem lautet ausformuliert: gibt es einen Algorithmus, der berechnet, ob ein Turingmaschinenprogramm in endlicher Zeit ein Ergebnis liefert? Turing konnte beweisen, dass man jeweils nur für spezielle Algorithmen zeigen kann, ob sie anhalten oder ob sie unendlich lange ausgeführt würden; und dass es sogar Turingmaschinenprogramme gibt, für die man nicht in endlicher Zeit eine Aussage darüber finden kann, ob dieses Programm anhält oder nicht. Es gibt keine allgemeine Lösung für das Halteproblem¹.

Auf dem Weg zu diesem Beweis fand er, dass mit einem speziellen Turingmaschinenprogramm, jede beliebige andere Turingmaschine (etwa mit mehreren Schreib-Lese-Köpfen) zusammen mit der Tabelle der Übergangsregeln simulierbar ist: er zeigte die Existenz eines universellen Turingmaschinenprogramms.

Roger Penrose² hat in seinem Buch „Computerdenken“, neben einem weiteren eleganten Beweis für die Unlösbarkeit des Halteproblems, dieses universelle Turingmaschinenprogramm explizit vorgestellt.

Bei der Untersuchung der von Turingmaschinen berechenbaren Funktionen, kann man feststellen, dass turingmaschinen-berechenbare Funktionen partiell rekursive Funktionen sind und dass umgekehrt jede partiell rekursive Funktion³ turingmaschinen-

¹ Turing, Alan M.: On Computable Numbers with an Application to the Entscheidungsproblem.

² Penrose, Roger: Computerdenken - Des Kaisers neue Kleider oder die Debatte um die Künstliche Intelligenz, Bewußtsein und die Gesetze der Physik.

³ Auf die Begriffe „berechenbar“ und „partiell rekursiv“ wird in dieser Diplomarbeit nicht näher eingegangen. Mehr dazu kann jedem Lehrbuch über Komplexitätstheorie entnommen werden.

berechenbar ist. Genau diese Aussage entspricht einer Formulierung der Churchschen These von Alonzo Church (1903-1995) die auch als Church-Turing-These bekannt ist: die partiell rekursiven Funktionen sind eine Präzisierung der intuitiven Berechenbarkeit⁴.

Die Turingmaschine ist nicht das einzige mathematische Modell, das die Eigenschaft hat, den partiell rekursiven Funktionen zu entsprechen. Gödel (1906-1978), Herbrand (1908-1931), Kleene (1909-1994) und Markov (1856-1922) haben ebenfalls verschiedene Ansätze verfolgt. Church selbst hat dazu das λ -Kalkül eingeführt, Emil Post (1897-1954) die nach ihm benannte Postsche Maschine. Die Register-Maschine von J.C. Shepherdson und H.E. Sturgis⁵ ist ein weiterer Ansatz. Alle diese Modelle sind äquivalent und daher existieren für alle Modelle jeweils universelle Programme. In der vorliegenden Arbeit wird untersucht, wie ein universelles Register-Maschinen-Programm erstellt werden kann.

Der Weg dahin besteht aus drei Schritten: zunächst wird das universelle Register-Maschinen-Programm als C++ Programm implementiert – ein Simulator für Register-Maschinen und Interpreter für Register-Maschinen-Programme in C++. Dieser Interpreter wird sich direkt aus der Arbeitsweise der Register-Maschine ergeben. Danach wird allgemein C++-Code in Register-Maschinen-Code umgewandelt, dies wird durch einen Compiler automatisiert. Durch Zusammenführen der beiden ersten Schritte wird der Interpreter durch den Compiler in Registemaschinencode übersetzt. Das Ergebnis dann ein Interpreter für Register-Maschinen-Programme in Register-Maschinen-Code: das universelle Register-Maschinen-Programm in Register-Maschinen-Code.

⁴ Church, Alonzo: An unsolvable problem in elementary number theory.

⁵ Shepherdson, J.C. & Sturgis, H.E.: Computability of Recursive Functions.

Namenskonvention und wichtige Anmerkungen

- In der vorliegenden Arbeit kommt an mehreren Stellen Programmcode vor. Dieser ist durch die nicht-proportionale Schriftart „Courier“ kenntlich gemacht. Beispiel:

```
printf("dies ist Programmcode\n");
```
- C++-Klassen werden am Anfang desjenigen Kapitels tabellarisch vorstellen, in dem sie verwendet werden. In den Tabellen werden jeweils nur die für die Methodik wichtigen Funktionen und Variablen aufgeführt und sie enthalten zusätzlich noch eine kurze Beschreibung der Klasse. Die Verwendung der Klasse wird im Laufe der Kapitel erklärt. In der Tabelle sind öffentliche Klassen-Methoden und Klassen-Eigenschaften unterstrichen dargestellt.
- In den C++ Quelltexten wurde fast durchgehend eine Namenskonvention eingehalten. Der Namenspräfix gibt dabei den Typ des Objekts an.

Präfix	Beschreibung	Beispiel
CTyp	Typ ist eine Klasse	CParse p;
STyp	Typ ist eine Struktur	SObjekt sMyObject;
m_Variable	Variable ist eine Membervariable einer Klasse	char * m_pcError;
pVariable	Variable ist ein Zeiger	long * pIVar;
sVariable	Variable hat als Typ eine Struktur	SObject sMyObject;
lVariable	Variable ist ein long-Wert	long lRegister;
rVariable	Variable ist ein CRegister-Wert	CRegister rProgram;

- Wenn von natürlichen Zahlen die Rede ist, so sind die natürlichen Zahlen, vereinigt mit der Menge, die die Zahl Null enthält, gemeint.
- Dieses Dokument, alle Programme und Programmquellen sind auf der beiliegenden CD enthalten.

II. Die Register-Maschine

Dieses Kapitel befasst sich mit dem Ursprung und den verschiedenen Definitionen der Register-Maschine. Zunächst werde ich auf die Register-Maschine, deren Definitionen und Programme eingehen, um danach ein C++-Programm vorzustellen, das eine Register-Maschine simuliert. Dieses Programm ist ein Interpreter für Register-Maschinen-Programme, ein universelles Register-Maschinen-Programm in C++.

1. Geschichte der Register-Maschine

Die Ansätze zur Untersuchung der Berechenbarkeit und allgemein den Eigenschaften von Algorithmen wurden von Turing, Church, Post u.a. bis etwa 1940 entwickelt. Aus deren Arbeiten entstanden dann die ersten theoretischen Modelle von Rechenmaschinen. Shepherdson und Sturgis haben 1961 zwei Probleme, speziell der Turingmaschine bemängelt:

- (a) Durch die Existenz nur eines Schreib-Lese-Kopfes müssen Berechnungen in kleinste Schritte aufgeteilt werden.
- (b) Es gibt nur ein Band, sodass die gesuchten Informationen weit voneinander entfernt sein können und Probleme dadurch entstehen, dass mehrere Informationseinheiten voneinander durch spezielle Zeichen oder Codierungen getrennt werden müssen.

Shepherdson und Sturgis haben versucht einen Mittelweg „between the practical and theoretical aspects of computation“⁶ zu beschreiten. Das Ergebnis ihrer Bemühungen war die erste Definition der Register-Maschine. Bevor diese erste Definition vorgestellt wird, folgt zunächst die in dieser Arbeit verwendete Definition,

2. Definition der Register-Maschine

Die Register-Maschine besteht aus endlich vielen Registern, die jeweils eine natürliche Zahl aufnehmen können. Man beachte, dass Register beliebig große natürliche Zahlen aufnehmen können - dies im Gegensatz zu realen Rechenmaschinen. Die Register sind eindeutig identifiziert durch ihren Index, das erste Register hat dabei den Index 1. Die Register-Maschine hat eine Kontrolleinheit, die eine Arbeitsanweisung enthält, diese besteht aus einer Folge von grundlegenden Befehlen. Diese Arbeitsanweisung wird von der Kontrolleinheit abgearbeitet – etwa um zwei Registerinhalte miteinander zu multiplizieren und das Ergebnis in einem dritten Register abzulegen. Die Register-Maschine stoppt, wenn das Programm komplett abgearbeitet wurde. Wie die Register-Maschine im Einzelnen arbeitet wird im nächsten Kapitel dargestellt.

⁶ Shepherdson, J.C. & Sturgis, H.E.: Computability of Recursive Functions. Seite 218.

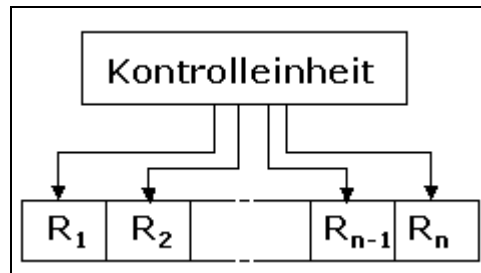


Abbildung II-1 Schema der Register-Maschine

Die Register-Maschine hat sich seit der ersten Definition durch Shepherdson und Sturgis stark gewandelt. Die Einteilung in Kontrolleinheit, Programm und Register ist zwar gleich geblieben, aber die Instruktionen, der Register-Maschinen-Code, wurde immer wieder verändert. Die Version, die in der vorliegenden Arbeit verwendet wird, ist die gleiche wie in der Vorlesung „Automaten- und Algorithmentheorie“ bei Prof. Schnitzspan, die dem Buch von Ottmann und Albert⁷ folgt. Zunächst werde ich diese Definition der Register-Maschine und des Register-Maschinen-Codes angeben, um danach auf die ursprüngliche, aber hier nicht verwendete zu kommen.

Definition der Register-Maschine

Sei $n \geq 1$ eine natürliche Zahl. Eine Register-Maschine mit n Registern besitzt n Register zur Speicherung beliebig großer natürlicher Zahlen. Die Register sind der Reihe nach geordnet, sodass man vom 1., 2., 3., ... , n -ten Register sprechen kann. Die Register können (programmgesteuert, schrittweise) verändert werden. Welche Änderungsmöglichkeiten es gibt, ist durch den Maschinen-Code festgelegt.

Definition des Register-Maschinen-Codes

Wir definieren induktiv gleichzeitig, was ein Register-Maschinen-Code und was die Wirkung eines solchen Befehlscodes ist (n ist dabei die Anzahl der Register):

1. Addition: A_i ist ein Register-Maschinen-Code ($1 \leq i \leq n$).
Wirkung: Es wird eine 1 zum Inhalt des i -ten Registers addiert.
2. Subtraktion: S_i ist ein Register-Maschinen-Code ($1 \leq i \leq n$).
Wirkung: Es wird eine 1 vom Inhalt des i -ten Registers subtrahiert, wenn dieser von Null verschieden ist, sonst bleibt der Inhalt unverändert.
3. Hintereinanderausführung: sind M_1 und M_2 Register-Maschinen-Codes, so auch M_1M_2 .
Wirkung: Es wird zunächst M_1 und anschließend M_2 ausgeführt.

⁷ Albert, J. und Ottmann, Th.: Automaten, Sprachen und Maschinen für Anwender.

4. Schleife: ist M ein Register-Maschinen-Code, so auch $(M)_i$ ($1 \leq i \leq n$).
Wirkung: Solange der Inhalt des i -ten Registers verschieden von Null ist, wird M ausgeführt.

Registermaschinencode

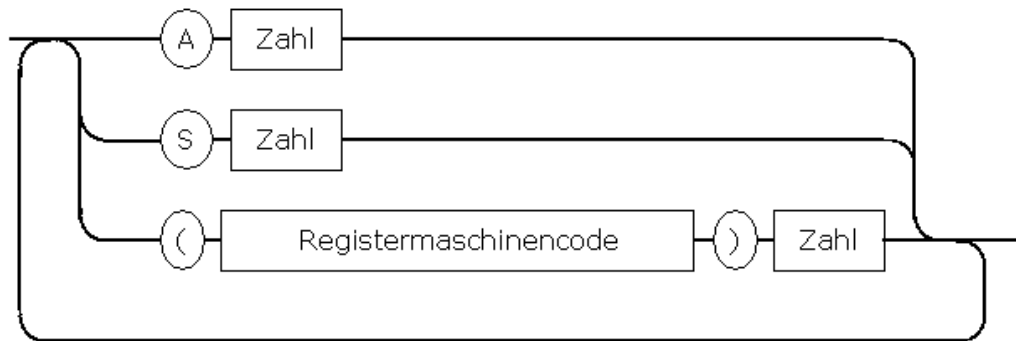


Abbildung II-2: Syntax des Register-Maschinen-Codes

Um die Wirkung der Befehlscodes zu verdeutlichen, möchte ich die sogenannte Compilersemantik angeben. Dabei wird allgemein Register-Maschinen-Code in C++-Code ausgedrückt. Um dies zu erreichen, benötigt man einen Datentyp für die Registerinhalte und ein Feld der Länge n für die Register. Den Datentyp der Register möchte ich "CRegister" (siehe Seite 20) nennen - später wird dieser Datentyp als eigenständige Klasse vorgestellt. Die Deklaration der Register kann dann wie folgt geschehen⁸:

```
CRegister R[n];
```

Mit der Abbildung P , die Register-Maschinen-Code in C++-Code wandelt, kann die Compilersemantik leicht angegeben werden zu:

1. $P(A_i): R[i]++$
2. $P(S_i): \text{if } (R[i] > 0) R[i]--$
3. $P(M_1M_2): \{ P(M_1); P(M_2); \}$
4. $P((M)_i): \text{while } (R[i] > 0) \{ P(M); \}$

Aus der Angabe der Compilersemantik lässt sich leicht ein Übersetzer von Register-Maschinen-Code nach C++-Code implementieren, etwa ein angepasstes Lex-Programm oder ein direkter Übersetzer.

⁸ Den Umstand, dass in C++ die Feldindizes mit 0 beginnen und bei $n-1$ enden möchte ich hier übergehen und so tun als ob die Indizes von 1 bis n laufen würden. Durch diese Entscheidung soll nur die Lesbarkeit erhöht werden.

Durch die Angabe der Compilersemantik wird auch die Wirkung der Schleife unter Punkt 4 der Definition von Register-Maschinen-Code (s.o.) klar. Dort war nicht definiert zu welchem Zeitpunkt ein Test auf "verschieden von Null" stattfinden soll (am Anfang, am Ende, immer). Hier habe ich mich für die gängige Lösung entschieden, die einer while-Schleife entspricht: Test auf „Registerinhalt nicht Null“ vor dem Schleifeninhalt durchführen.

Zur Angabe eines Register-Maschinen-Programms gehört, neben dem Register-Maschinen-Code, auch der Inhalt der n Register⁹. Dieser Inhalt wird hinter dem Register-Maschinen-Code angegeben. Ein Punkt trennt dabei Register-Maschinen-Code und Registerinhalt: $\langle Code \rangle . \langle Inhalt\ Register\ 1 \rangle, \langle Inhalt\ Register\ 2 \rangle, \dots, \langle Inhalt\ Register\ n \rangle$. Dabei werden allerdings nicht alle n Register aufgelistet, sondern nur die ersten relevanten Register; nicht angegebene Registerinhalte sind immer mit Null belegt.

Zur Verdeutlichung der Wirkung von Register-Maschinen-Programmen nun drei Beispiele.

- Leeren des Inhalts eines Registers; hier Register 1: „ $(S_1)_1 . 5, 7, 0$ “ wird nach Ausführung zu „ $. 0, 7, 0$ “
- Transferieren des Inhalts eines Registers; hier von Register 1 nach Register 2: „ $(S_1 A_2)_1 . 5, 0, 3, 0$ “ wird nach Ausführung zu „ $. 0, 5, 3, 0$ “
- Kopieren des Inhalts eines Registers; hier von Register 1 nach Register 2 (dabei ist Register 4 das benötigte Hilfsregister): „ $(S_1 A_2 A_4)_1 (S_4 A_1)_4 . 5, 0, 3, 0$ “ wird nach Ausführung zu „ $. 5, 5, 3, 0$ “

⁹ Ob die Registerinhalte zum Register-Maschinen-Programm gehören, ist eine Definitionsfrage; ich habe mich aus praktischen Erwägungen dafür entschieden u.a. auch deswegen, weil der weiter unten vorgestellte Compiler z.T. auch Programmkonstanten in den Registerinhalten ablegt.

3. Arbeitsweise der Register-Maschine

Die Register-Maschine kann Register-Maschinen-Programme bearbeiten. Durch die Compilersemantik ist klar geworden, wie der Code der Register-Maschine zu verstehen ist. Jetzt kann daraus die genaue Beschreibung der Arbeitsweise der Register-Maschine im Einzelnen folgen. Ich werde nun eine Maschine vorstellen, die später durch den „Interpreter“ repräsentiert wird. Die generelle Vorgehensweise der Maschine ist dabei:

1. Prüfe, ob Programmcode vorhanden ist, falls kein Programmcode vorhanden ist, gehe zu Schritt 4
2. Führe nächsten Programmschritt aus
3. Gehe zu Schritt 1
4. Programmende

Zunächst werde ich auf Punkt 2 eingehen, die Ausführung eines einzelnen Programmschrittes. Dazu gibt es in der Vorlesung „Automaten- und Algorithmentheorie“ von Prof. Schnitzspan¹⁰ eine Funktion g , die dies beschreibt. Diese Funktion erhält als Parameter das aktuelle Register-Maschinen-Programm, hier aufgeteilt in Befehlscode (auch „Stapel“) und Registerinhalte (auch „Tupel“). Die Funktion liefert dann das neue Register-Maschinen-Programm (den neuen Befehlscode und die neuen Registerinhalte). Die Funktion g , die aus altem Stapel und altem Tupel den neuen Stapel und das neue Tupel berechnet lautet:

alt		Bedingung	neu	
Stapel	Tupel		Stapel	Tupel
$A_i M$	x_1, x_2, x_3, \dots	<i>keine</i>	M	$x_1, x_2, \dots, x_{i-1}, x_i + 1, x_{i+1}, \dots$
$S_i M$	x_1, x_2, x_3, \dots	$x_i = 0$	M	$x_1, x_2, \dots, x_{i-1}, 0, x_{i+1}, \dots$
		$x_i \neq 0$	M	$x_1, x_2, \dots, x_{i-1}, x_i - 1, x_{i+1}, \dots$
$(M_1)_i M_2$	x_1, x_2, x_3, \dots	$x_i = 0$	M_2	x_1, x_2, x_3, \dots
		$x_i \neq 0$	$M_1 (M_1)_i M_2$	x_1, x_2, x_3, \dots

Tabelle II-1: Die Funktion g , die aus altem Stapel und altem Tupel den neuen Stapel und das neue Tupel berechnet

Um die Arbeitsweise zu veranschaulichen, hier zunächst ein ausführliches Beispiel. Es soll Register 1 und Register 2 auf das Register 3 addiert werden. Jeder Ausführungsschritt ist hierbei aufgelistet:

¹⁰ Schnitzspan, H.: Vorlesung AAT.

1. $(A_3 A_4 S_1)_1 (A_1 S_4)_4 (A_3 A_4 S_2)_2 (A_2 S_4)_4 \cdot 5,7$
2. $A_3 A_4 S_1 (A_3 A_4 S_1)_1 (A_1 S_4)_4 (A_3 A_4 S_2)_2 (A_2 S_4)_4 \cdot 5,7$
3. $A_4 S_1 (A_3 A_4 S_1)_1 (A_1 S_4)_4 (A_3 A_4 S_2)_2 (A_2 S_4)_4 \cdot 5,7,1$
4. $S_1 (A_3 A_4 S_1)_1 (A_1 S_4)_4 (A_3 A_4 S_2)_2 (A_2 S_4)_4 \cdot 5,7,1,1$
5. $(A_3 A_4 S_1)_1 (A_1 S_4)_4 (A_3 A_4 S_2)_2 (A_2 S_4)_4 \cdot 4,7,1,1$
6. $A_3 A_4 S_1 (A_3 A_4 S_1)_1 (A_1 S_4)_4 (A_3 A_4 S_2)_2 (A_2 S_4)_4 \cdot 4,7,1,1$
7. $A_4 S_1 (A_3 A_4 S_1)_1 (A_1 S_4)_4 (A_3 A_4 S_2)_2 (A_2 S_4)_4 \cdot 4,7,2,1$
8. $S_1 (A_3 A_4 S_1)_1 (A_1 S_4)_4 (A_3 A_4 S_2)_2 (A_2 S_4)_4 \cdot 4,7,2,2$
9. $(A_3 A_4 S_1)_1 (A_1 S_4)_4 (A_3 A_4 S_2)_2 (A_2 S_4)_4 \cdot 3,7,2,2$
10. $A_3 A_4 S_1 (A_3 A_4 S_1)_1 (A_1 S_4)_4 (A_3 A_4 S_2)_2 (A_2 S_4)_4 \cdot 3,7,2,2$
11. $A_4 S_1 (A_3 A_4 S_1)_1 (A_1 S_4)_4 (A_3 A_4 S_2)_2 (A_2 S_4)_4 \cdot 3,7,3,2$
12. $S_1 (A_3 A_4 S_1)_1 (A_1 S_4)_4 (A_3 A_4 S_2)_2 (A_2 S_4)_4 \cdot 3,7,3,3$
13. $(A_3 A_4 S_1)_1 (A_1 S_4)_4 (A_3 A_4 S_2)_2 (A_2 S_4)_4 \cdot 2,7,3,3$
14. $A_3 A_4 S_1 (A_3 A_4 S_1)_1 (A_1 S_4)_4 (A_3 A_4 S_2)_2 (A_2 S_4)_4 \cdot 2,7,3,3$
15. $A_4 S_1 (A_3 A_4 S_1)_1 (A_1 S_4)_4 (A_3 A_4 S_2)_2 (A_2 S_4)_4 \cdot 2,7,4,3$
16. $S_1 (A_3 A_4 S_1)_1 (A_1 S_4)_4 (A_3 A_4 S_2)_2 (A_2 S_4)_4 \cdot 2,7,4,4$
17. $(A_3 A_4 S_1)_1 (A_1 S_4)_4 (A_3 A_4 S_2)_2 (A_2 S_4)_4 \cdot 1,7,4,4$
18. $A_3 A_4 S_1 (A_3 A_4 S_1)_1 (A_1 S_4)_4 (A_3 A_4 S_2)_2 (A_2 S_4)_4 \cdot 1,7,4,4$
19. $A_4 S_1 (A_3 A_4 S_1)_1 (A_1 S_4)_4 (A_3 A_4 S_2)_2 (A_2 S_4)_4 \cdot 1,7,5,4$
20. $S_1 (A_3 A_4 S_1)_1 (A_1 S_4)_4 (A_3 A_4 S_2)_2 (A_2 S_4)_4 \cdot 1,7,5,5$
21. $(A_3 A_4 S_1)_1 (A_1 S_4)_4 (A_3 A_4 S_2)_2 (A_2 S_4)_4 \cdot 0,7,5,5$
22. $(A_1 S_4)_4 (A_3 A_4 S_2)_2 (A_2 S_4)_4 \cdot 0,7,5,5$
23. $A_1 S_4 (A_1 S_4)_4 (A_3 A_4 S_2)_2 (A_2 S_4)_4 \cdot 0,7,5,5$
24. $S_4 (A_1 S_4)_4 (A_3 A_4 S_2)_2 (A_2 S_4)_4 \cdot 1,7,5,5$
25. $(A_1 S_4)_4 (A_3 A_4 S_2)_2 (A_2 S_4)_4 \cdot 1,7,5,4$
26. $A_1 S_4 (A_1 S_4)_4 (A_3 A_4 S_2)_2 (A_2 S_4)_4 \cdot 1,7,5,4$
27. $S_4 (A_1 S_4)_4 (A_3 A_4 S_2)_2 (A_2 S_4)_4 \cdot 2,7,5,4$
28. $(A_1 S_4)_4 (A_3 A_4 S_2)_2 (A_2 S_4)_4 \cdot 2,7,5,3$
29. $A_1 S_4 (A_1 S_4)_4 (A_3 A_4 S_2)_2 (A_2 S_4)_4 \cdot 2,7,5,3$
30. $S_4 (A_1 S_4)_4 (A_3 A_4 S_2)_2 (A_2 S_4)_4 \cdot 3,7,5,3$
31. $(A_1 S_4)_4 (A_3 A_4 S_2)_2 (A_2 S_4)_4 \cdot 3,7,5,2$
32. $A_1 S_4 (A_1 S_4)_4 (A_3 A_4 S_2)_2 (A_2 S_4)_4 \cdot 3,7,5,2$
33. $S_4 (A_1 S_4)_4 (A_3 A_4 S_2)_2 (A_2 S_4)_4 \cdot 4,7,5,2$
34. $(A_1 S_4)_4 (A_3 A_4 S_2)_2 (A_2 S_4)_4 \cdot 4,7,5,1$
35. $A_1 S_4 (A_1 S_4)_4 (A_3 A_4 S_2)_2 (A_2 S_4)_4 \cdot 4,7,5,1$
36. $S_4 (A_1 S_4)_4 (A_3 A_4 S_2)_2 (A_2 S_4)_4 \cdot 5,7,5,1$
37. $(A_1 S_4)_4 (A_3 A_4 S_2)_2 (A_2 S_4)_4 \cdot 5,7,5$
38. $(A_3 A_4 S_2)_2 (A_2 S_4)_4 \cdot 5,7,5$
39. $A_3 A_4 S_2 (A_3 A_4 S_2)_2 (A_2 S_4)_4 \cdot 5,7,5$
40. $A_4 S_2 (A_3 A_4 S_2)_2 (A_2 S_4)_4 \cdot 5,7,6$
41. $S_2 (A_3 A_4 S_2)_2 (A_2 S_4)_4 \cdot 5,7,6,1$
42. $(A_3 A_4 S_2)_2 (A_2 S_4)_4 \cdot 5,6,6,1$
43. $A_3 A_4 S_2 (A_3 A_4 S_2)_2 (A_2 S_4)_4 \cdot 5,6,6,1$
44. $A_4 S_2 (A_3 A_4 S_2)_2 (A_2 S_4)_4 \cdot 5,6,7,1$
45. $S_2 (A_3 A_4 S_2)_2 (A_2 S_4)_4 \cdot 5,6,7,2$
46. $(A_3 A_4 S_2)_2 (A_2 S_4)_4 \cdot 5,5,7,2$
47. $A_3 A_4 S_2 (A_3 A_4 S_2)_2 (A_2 S_4)_4 \cdot 5,5,7,2$
48. $A_4 S_2 (A_3 A_4 S_2)_2 (A_2 S_4)_4 \cdot 5,5,8,2$
49. $S_2 (A_3 A_4 S_2)_2 (A_2 S_4)_4 \cdot 5,5,8,3$
50. $(A_3 A_4 S_2)_2 (A_2 S_4)_4 \cdot 5,4,8,3$
51. $A_3 A_4 S_2 (A_3 A_4 S_2)_2 (A_2 S_4)_4 \cdot 5,4,8,3$
52. $A_4 S_2 (A_3 A_4 S_2)_2 (A_2 S_4)_4 \cdot 5,4,9,3$
53. $S_2 (A_3 A_4 S_2)_2 (A_2 S_4)_4 \cdot 5,4,9,4$
54. $(A_3 A_4 S_2)_2 (A_2 S_4)_4 \cdot 5,3,9,4$
55. $A_3 A_4 S_2 (A_3 A_4 S_2)_2 (A_2 S_4)_4 \cdot 5,3,9,4$
56. $A_4 S_2 (A_3 A_4 S_2)_2 (A_2 S_4)_4 \cdot 5,3,10,4$
57. $S_2 (A_3 A_4 S_2)_2 (A_2 S_4)_4 \cdot 5,3,10,5$
58. $(A_3 A_4 S_2)_2 (A_2 S_4)_4 \cdot 5,2,10,5$
59. $A_3 A_4 S_2 (A_3 A_4 S_2)_2 (A_2 S_4)_4 \cdot 5,2,10,5$
60. $A_4 S_2 (A_3 A_4 S_2)_2 (A_2 S_4)_4 \cdot 5,2,11,5$
61. $S_2 (A_3 A_4 S_2)_2 (A_2 S_4)_4 \cdot 5,2,11,6$
62. $(A_3 A_4 S_2)_2 (A_2 S_4)_4 \cdot 5,1,11,6$
63. $A_3 A_4 S_2 (A_3 A_4 S_2)_2 (A_2 S_4)_4 \cdot 5,1,11,6$
64. $A_4 S_2 (A_3 A_4 S_2)_2 (A_2 S_4)_4 \cdot 5,1,12,6$
65. $S_2 (A_3 A_4 S_2)_2 (A_2 S_4)_4 \cdot 5,1,12,7$
66. $(A_3 A_4 S_2)_2 (A_2 S_4)_4 \cdot 5,0,12,7$
67. $(A_2 S_4)_4 \cdot 5,0,12,7$
68. $A_2 S_4 (A_2 S_4)_4 \cdot 5,0,12,7$
69. $S_4 (A_2 S_4)_4 \cdot 5,1,12,7$
70. $(A_2 S_4)_4 \cdot 5,1,12,6$
71. $A_2 S_4 (A_2 S_4)_4 \cdot 5,1,12,6$
72. $S_4 (A_2 S_4)_4 \cdot 5,2,12,6$
73. $(A_2 S_4)_4 \cdot 5,2,12,5$
74. $A_2 S_4 (A_2 S_4)_4 \cdot 5,2,12,5$
75. $S_4 (A_2 S_4)_4 \cdot 5,3,12,5$
76. $(A_2 S_4)_4 \cdot 5,3,12,4$
77. $A_2 S_4 (A_2 S_4)_4 \cdot 5,3,12,4$
78. $S_4 (A_2 S_4)_4 \cdot 5,4,12,4$
79. $(A_2 S_4)_4 \cdot 5,4,12,3$
80. $A_2 S_4 (A_2 S_4)_4 \cdot 5,4,12,3$
81. $S_4 (A_2 S_4)_4 \cdot 5,5,12,3$
82. $(A_2 S_4)_4 \cdot 5,5,12,2$
83. $A_2 S_4 (A_2 S_4)_4 \cdot 5,5,12,2$
84. $S_4 (A_2 S_4)_4 \cdot 5,6,12,2$
85. $(A_2 S_4)_4 \cdot 5,6,12,1$
86. $A_2 S_4 (A_2 S_4)_4 \cdot 5,6,12,1$
87. $S_4 (A_2 S_4)_4 \cdot 5,7,12,1$
88. $(A_2 S_4)_4 \cdot 5,7,12$
89. $5,7,12$

Ziel dieser Diplomarbeit ist das Erstellen eines universellen Register-Maschinen-Programms. Um dies zu erreichen, muss ein Register-Maschinen-Programm in der Lage sein, ein zweites Register-Maschinen-Programm abzuarbeiten. Dies bedingt, dass das zu simulierende Register-Maschinen-Programm als eine oder mehrere natürliche Zahlen codiert ist, da das Programm in Registern abgelegt werden muss. Das zu simulierende Register-Maschinen-Programm¹¹ (M, \bar{x}) wird dann mit der Codierung Γ zu (μ, χ) , wobei

¹¹ Um die Schreibweise einfach zu halten, ist hier als Kurzform für x_1, \dots, x_n der Vektor \bar{x} gewählt worden, er steht für die Registerinhalte. M ist Register-Maschinen-Code, also eine Folge von Register-Maschinen-Befehlen.

nicht festgelegt sein muss, ob μ oder χ eine einzige natürliche Zahl oder ein Vektor ist. Es muss also gelten: $\Gamma(M, \bar{x}) = (\mu, \chi)$, dabei soll Γ eine bijektive Funktion sein, sodass wiederum $\Gamma^{-1}(\Gamma(M, \bar{x})) = (M, \bar{x})$ gilt.

Definition eines universellen Register-Maschinen-Programms:

Ein Register-Maschinen-Programm U heißt universell, wenn für beliebige $n \geq 1$, ein beliebiges Programm M einer Register-Maschine mit n Registern und beliebige Anfangsregisterinhalte x_1, \dots, x_n gilt:

1. M hält angesetzt auf x_1, \dots, x_n genau dann, wenn U angesetzt auf $\mu, \chi, 0$ (nach endlich vielen Schritten) hält.
2. Wenn M angesetzt auf x_1, \dots, x_n nach endlich vielen Schritten hält, so gilt:

$$M . x_1, \dots, x_n \rightarrow . y_1, \dots, y_n$$

genau dann, wenn

$$U . \mu, \chi, 0 \rightarrow . 0, \psi, 0$$

In der vorliegenden Arbeit wird μ , der Register-Maschinen-Code, sowie χ bzw. ψ als jeweils eine einzige natürliche Zahl codiert sein. (Später wird diese spezielle bijektive Codierung vorgestellt (Seite 23).) Dass der Register-Maschinen-Code als einzelne Zahl codiert wird, liegt daran, dass bei der Abarbeitung dann immer nur mit dieser Zahl gearbeitet werden muss. Die Registerinhalte sind als eine einzige Zahl codiert, da eine Aufteilung in eine feste Anzahl nicht praktikabel wäre. Eine Abbildung „simuliertes Register $i \rightarrow$ Register j “ etwa durch eine feste Verschiebung um m Register, kann bei der gegebenen Register-Maschine nicht erfolgen. Jede simulierte Operation auf ein Register muss die äquivalente Operation in der ausführenden Maschine zur Folge haben: $\Gamma(A_i, \bar{x}) = (\alpha_j, \chi)$. Da es allerdings keine relativen Zugriffsfunktionen auf Register gibt, sondern nur absolute, muss die Übersetzung einer relativen Operation durch einen Satz von Abfragen geregelt sein. Die Anzahl dieser Abfragen sind bei n Registern ebenfalls n . Da aber n beliebig und nicht begrenzt sein soll, ist diese Methode unpraktikabel.

Den Beweis der Existenz des universellen Register-Maschinen-Programms explizit anzugeben, würde über den Rahmen dieser Diplomarbeit hinausgehen, kann aber bei Cutland¹² nachgelesen werden.

4. Andere Definitionen der Register-Maschine

Die älteste Definition der Register-Maschine ist von 1961 und stammt von Shepherdson und Sturgis. Dabei ist die Einteilung der Register-Maschine in Kontrolleinheit, Code

¹² Cutland, N.: Computability: an introduction to recursive function theory. Seite 95.

und Register identisch mit der bereits gegebenen – der Register-Maschinen-Code und die Bearbeitung sind etwas anders:

Definition der Register-Maschinen-Befehle nach Shepherdson und Sturgis¹³ für eine Register-Maschine mit unendlich vielen Registern und m Register-Maschinen-Befehlen:

1. Addition: P(i) ist ein Register-Maschinen-Befehl ($1 \leq i$).
Wirkung: Es wird eine 1 zum Inhalt des i-ten Registers addiert.
2. Subtraktion: D(i) ist ein Register-Maschinen-Befehl ($1 \leq i$).
Wirkung: Es wird eine 1 vom Inhalt des i-ten Registers subtrahiert, wenn dieser von Null verschieden ist, sonst bleibt der Inhalt unverändert.
3. Löschen: O(i) ist ein Register-Maschinen-Befehl ($1 \leq i$).
Wirkung: Der Inhalt des Registers i wird durch Null ersetzt.
4. Kopieren: C(i,j) ist ein Register-Maschinen-Befehl ($1 \leq i$ und $1 \leq j \leq n$).
Wirkung: Der Inhalt von Register i wird in das Register j kopiert, der Inhalt des Registers j wird überschrieben.
5. Unbedingter Sprung: J[p] ist ein Register-Maschinen-Befehl ($1 \leq p \leq m+1$).
Wirkung: Statt den nächsten Register-Maschinen-Befehl zu bearbeiten, bearbeite den p-ten.
6. Bedingter Sprung: J(i) [p] ist ein Register-Maschinen-Befehl ($1 \leq i$ und $1 \leq p \leq m+1$).
Wirkung: Falls Inhalt Register i gleich Null ist, bearbeite statt des nächsten Register-Maschinen-Befehls den p-ten Register-Maschinen-Befehl.

Dabei sind alle Register-Maschinen-Befehle der Reihe nach durchnummeriert (1 bis m) und ein Register-Maschinen-Programm sieht dann beispielsweise so aus:

1. J(1) [5]
2. D(1)
3. P(2)
4. J[1]

Dieses Shepherdson-Sturgis-Register-Maschinen-Programm ist dem Register-Maschinen-Programm $(S_1 A_2)_1$ äquivalent. Der bedingte Sprung nach Register-Maschinen-Befehl 5 entspricht dabei dem Abbruch.

Auffallendster Unterschied zur Definition von Albert und Ottman ist die Verwendung des Sprung-Befehls, dessen Aufgabe durch die bedingte Schleife bei Albert und Ottmann übernommen werden kann. Die 1961 bereits vorhandenen ersten Rechenanlagen, die mit Assembler programmiert wurden, verstanden ebenfalls Befehle mit bedingten und unbedingten Sprüngen.

¹³ Shepherdson, J.C. & Sturgis, H.E.: Computability of Recursive Functions. Seite 219.

Eine weitere Besonderheit ist die Verwendung von unendlich vielen Registern (bei Shepherdson und Sturgis wird die Register-Maschine je nach Anzahl der Register bezeichnet „URM“: unlimited register machine, „LRM“: limited register machine, „SRM“: single register machine). Es ist aber keinesfalls eine Einschränkung in der von uns verwendeten Register-Maschine¹⁴, da zum einen die Referenzen auf die Registerinhalte (in den Definitionen jeweils „i“ bzw. „j“) während des Programmlaufs nicht verändert werden können und damit vor dem Programmstart feststeht, wie viele Register benötigt werden. Zum anderen werden wir später sehen, dass mit einer Register-Maschine mit begrenzter Registerzahl eine Register-Maschine mit unendlicher Registerzahl simuliert werden kann.

Ein wichtiges Ergebnis der Arbeit von Shepherdson und Sturgis ist, dass die Register-Maschinen-Befehle reduziert werden können auf genau drei:

Reduzierte Register-Maschinen-Befehle nach Shepherdson und Sturgis:

1. Addition: $P(i)$ ist ein Register-Maschinen-Befehl ($1 \leq i$).
Wirkung: Es wird eine 1 zum Inhalt des i -ten Registers addiert.
2. Subtraktion: $D(i)$ ist ein Register-Maschinen-Befehl ($1 \leq i$).
Wirkung: Es wird eine 1 vom Inhalt des i -ten Registers subtrahiert, wenn dieser von Null verschieden ist, sonst bleibt der Inhalt unverändert.
3. Bedingter Sprung: $J(i) [p]$ ist ein Register-Maschinen-Befehl ($1 \leq i$ und $1 \leq p \leq m+1$).
Wirkung: Falls Inhalt Register i gleich Null ist, bearbeite statt des nächsten Register-Maschinen-Befehls den p -ten Register-Maschinen-Befehl.

Damit haben wir die gleiche Anzahl an Befehlen wie in der Register-Maschine von Ottmann und Albert.

Auf dem Ansatz von Shepherdson und Sturgis aufbauend wählte Cutland 1980 die Register-Maschinen-Befehle für eine Register-Maschine für unendlich viele Register wie folgt:

Register-Maschinen-Befehle nach Cutland¹⁵:

1. Initialisierung: $Z(i)$ ist ein Register-Maschinen-Befehl ($1 \leq i$).
Wirkung: Der Inhalt des i -ten Registers wird auf Null geändert.
2. Nachfolger: $S(i)$ ist ein Register-Maschinen-Befehl ($1 \leq i$).
Wirkung: Zum Inhalt des i -ten Registers wird 1 addiert.
3. Transfer: $T(i,j)$ ist ein Register-Maschinen-Befehl ($1 \leq i, 1 \leq j$).
Wirkung: Der Inhalt des Registers j wird durch den Inhalt von Register i ersetzt. Der Inhalt von Register i bleibt unverändert.

¹⁴ Voraussetzung dafür ist, dass die Anzahl der Register genügend groß gewählt wurde.

¹⁵ Cutland, N.: Computability: an introduction to recursive function theory. Seite 10.

4. Bedingter Sprung: $J(i,j,p)$ ist ein Register-Maschinen-Befehl ($1 \leq i, 1 \leq j$ und $1 \leq p$).
Wirkung: Falls Register i den gleichen Inhalt hat wie Register j , bearbeite statt des nächsten Register-Maschinen-Befehls den p -ten Register-Maschinen-Befehl.

Es ist leicht einzusehen, dass das Quadrupel Initialisierung/Nachfolger/Transfer/Sprung durch das Paar Addition/Subtraktion/Schleife ersetzt werden kann.

5. Der Interpreter von Register-Maschinen-Programmen

Aufgabe dieses Kapitels ist es eine Implementierung der virtuellen Maschine „Register-Maschine“ zu beschreiben. Die theoretische Arbeitsweise und die Einzelteile wie „Register“, „Stapel“, „Tupel“ wurden definiert, diese müssen nun in C++ Code überführt werden.

5.1. Die Vorbereitungen

Um mit unendlich großen natürlichen Zahlen umgehen zu können, wurde eine eigene C++-Klasse erstellt. Die Klasse CRegister kann alle gängigen Operationen wie Addieren, Subtrahieren, Multiplizieren, Dividieren und Modulo mit anderen CRegister-Zahlen oder long-Zahlen (siehe Glossar Seite 53) durchführen. Diese Klasse bildet den Datentyp für die Registerinhalte der Register-Maschine. (Die komplette Klassendefinition sowie eine Übersichtstabelle kann dem Anhang auf Seite 56 entnommen werden.)

CRegister	
Klasse für große natürliche Zahlen. Stellt alle gängigen Rechenoperationen zur Verfügung.	
Methode/ Eigenschaft	Beschreibung
CRegister	Konstruktor
~CRegister	Destruktor
operator=	Kopiert das gegebene Register auf das aktuelle, sodass deren Werte gleich sind
operator+	Addition von zwei Registern: $R_{\text{this}} + R_x$, liefert das Ergebnis als neues Register zurück
operator-	Subtraktion von zwei Registern: $R_{\text{this}} - R_x$, liefert das Ergebnis als neues Register zurück
operator++	Inkrementiert das aktuelle Register (Addition von 1)
operator--	Dekrementiert das aktuelle Register (Subtraktion von 1)
operator+=	Addiert das übergebene Register zum aktuellen hinzu
operator-=	Subtrahiert das übergebene Register vom aktuellen

Tabelle II-2: Klasse CRegister

Um die Implementierung einfach zu halten, wurde von vornherein darauf verzichtet negative Zahlen darzustellen. Jede Subtraktion „ $n - m$ “ mit $m \geq n$ und $m, n \geq 0$ ergibt Null.

5.2. Der Interpreter

Der Interpreter für Register-Maschinen-Programme ist in C++ implementiert. Er arbeitet streng nach der Funktion g (siehe Seite 14), d.h. es gibt eine C++ Funktion, die der Funktion g entspricht, die genau einen Schritt in der Verarbeitung des Register-Maschinen-Programms durchführt. Die Funktion g wird so lange ausgeführt, bis der Programmstapel leer ist.

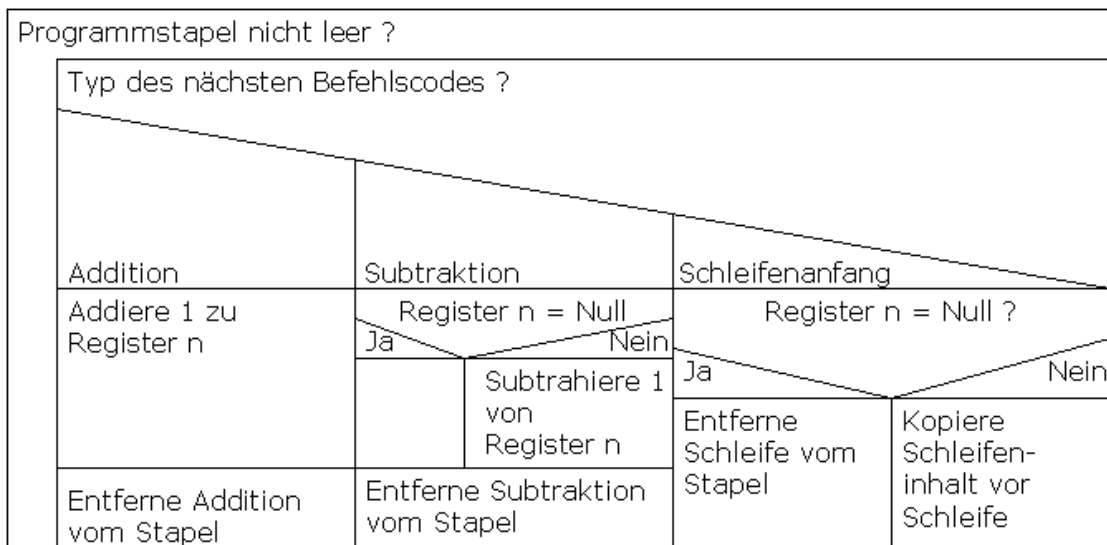


Abbildung II-3: Schema des Interpreters

Diesen Interpreter habe ich implementiert, er ist im Anhang angegeben. Er benutzt dabei eine Zeichenkette als Ablage des Register-Maschinen-Codes und ein Feld des Typs CRegister als Ablage der Registerinhalte. Es gibt drei Funktionen, für die Bearbeitung der Addition, Subtraktion und der Schleife. Dazu noch eine Funktion die diese drei Funktionen umschließt (also der Funktion g entspricht) und die in einer Schleife so oft durchlaufen wird, bis das Programm vollständig bearbeitet wurde.

Der Interpreter, der nach dem obigen Schema arbeitet, ist für ein größeres Programm (>15 kByte, dies entspricht etwa 5000 Instruktionen) viel zu langsam; er schafft in der optimiert compilierten Version etwa 30000 Schritte pro Sekunde auf einem Celeron 466. Schon das Addieren eines Registerinhaltes zu einem zweiten Register in einer Schleife wie bei $(S_1 A_2)_1$ besteht aus drei Einzelschritten: 1. Schleifentest, 2. Subtraktion, 3. Addition. Ist der Registerinhalt 10000, so dauert die Ausführung dieses einfachen Register-Maschinen-Programms eine Sekunde.

Der größte Anteil in Register-Maschinen-Programmen besteht darin, ein Quellregister in ein oder mehrere Zielregister zu kopieren. Um dies zu beschleunigen, wurden einige Verbesserungen oder „Abkürzungen“ eingebaut, sodass ganze Befehls-Sequenzen in einem Schritt ausgeführt werden können.

1. $(S_i)_i$ wird sofort ausgeführt, Register i wird auf Null gesetzt.

2. $(S_i A_j)_i$ wird sofort ausgeführt, zu Register j wird der Inhalt von Register i addiert und das Register i auf Null gesetzt.
3. $(S_i A_j A_k)_i$ wird sofort ausgeführt, zu Register j und Register k wird der Inhalt von Register i addiert und das Register i auf Null gesetzt.
4. $(S_i A_j S_k)_i$ wird sofort ausgeführt, zu Register j wird der Inhalt von Register i addiert, von Register k wird der Inhalt von Register i subtrahiert und das Register i auf Null gesetzt.
5. $(S_i S_j S_k)_i$ wird sofort ausgeführt, von Register j und Register k wird der Inhalt von Register i subtrahiert und das Register i auf Null gesetzt.

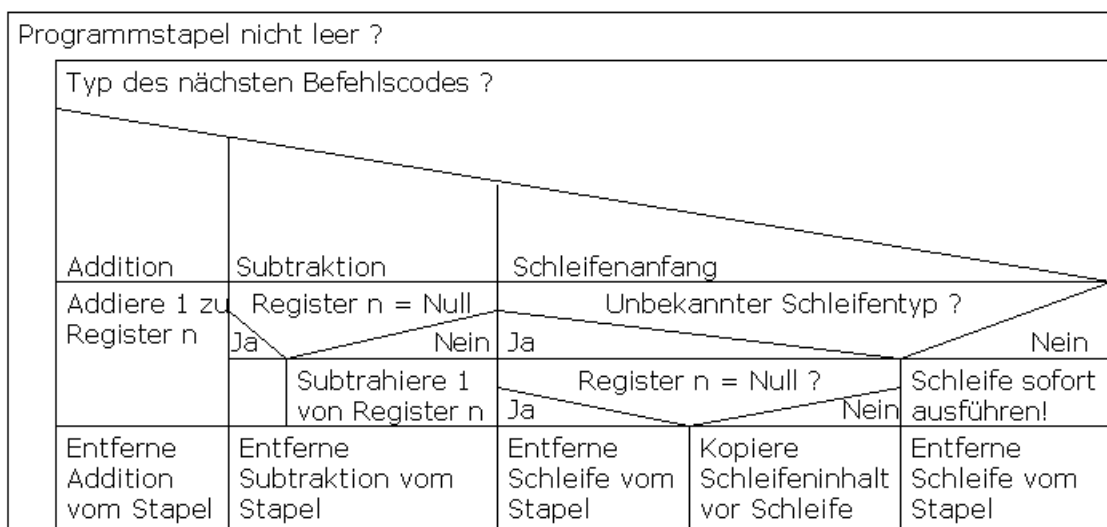


Abbildung II-4: Schema des optimierten Interpreters

Damit kann das Beispiel von oben $(S_1 A_3 A_4)_1 (S_4 A_1)_4 (S_2 A_3 A_4)_2 (S_4 A_2)_4 \cdot 5, 7$ in 4 Schritten abgearbeitet werden:

1. $(S_1 A_3 A_4)_1 (S_4 A_1)_4 (S_2 A_3 A_4)_2 (S_4 A_2)_4 \cdot 5, 7$
2. $(S_4 A_1)_4 (S_2 A_3 A_4)_2 (S_4 A_2)_4 \cdot 0, 7, 5, 5$
3. $(S_2 A_3 A_4)_2 (S_4 A_2)_4 \cdot 5, 7, 5$
4. $(S_4 A_2)_4 \cdot 5, 0, 12, 7$
5. $\cdot 5, 7, 12$

Der Interpreter ist als einfaches Konsolenprogramm ohne grafische Oberfläche, nur mit Textein/ausgabe, in C++ implementiert. Hier ein Bildschirmausdruck für das o.g. Register-Maschinen-Programm:

```

C:\Users\Harry\vcpp\D05\Release\D05.exe
Interpreter für die Registermaschine <5>
*****
<0/46> <S1 A3 A4 >1 <S4 A1 >4 <S2 A3 A4 >2 <S4 A2 >4 . 5,7
Programmstart...
Eingabe: <nnn[c][d]> mit nnn = Anzahl der Durchläufe ohne Unterbrechung
           [c] : 'c' = kein Befehlscode, 'C' = Befehlscode anzeigen
           [d] : 'd' = keine Daten, 'D' = Daten anzeigen

>5DCL

<20/34> <S4 A1 >4 <S2 A3 A4 >2 <S4 A2 >4 . 0,7,5,5
<35/24> <S2 A3 A4 >2 <S4 A2 >4 . 5,7,5
<63/11> <S4 A2 >4 . 5,0,12,7
<84/1> . 5,7,12
<84/1> . 5,7,12
*****
<84/1> . 5,7,12

```

Abbildung II-5: Bildschirmausdruck des Laufzeitoptimierten Interpreters

Einen Interpreter für Register-Maschinen-Code in Register-Maschinen-Code zu erstellen ist das eigentliche Ziel dieser Arbeit; dies kann aber ohne generelle Umstrukturierung des vorliegenden Interpreters, u.a. wegen der Benutzung von Zeichenketten und der eingebauten Optimierung, nicht geschehen. Dazu müsste eine Übersetzung der Zeichenkettenoperationen durch Register-Maschinen-Code erfolgen. Das folgende Kapitel beschreibt einen weiteren Interpreter, der mit nur einem Typ von Variablen implementiert ist und einfachste Funktionsaufrufe und Berechnungen enthält.

5.3. Der Interpreter II

Eine wichtige Nebenbedingung für die Implementierung des Interpreters war es, für die Verarbeitung des Programmcodes und der Registerinhalte nur Variablen vom Typ CRegister zu verwenden, damit der Interpreter später in ein Register-Maschinen-Programm umgewandelt werden kann. Dies wurde dadurch erreicht, dass Programmcode und Registerinhalt in jeweils einer CRegister-Variable gehalten werden. Dies wird bei Ottmann und Albert vorgeschlagen und ist ähnlich für das universelle Turingmaschinenprogramm schon von Penrose durchgeführt worden.

Diese Methode erlaubt auch „Adress-Relative“ Operationen durchzuführen: z.B. muss der Schleifeninhalt bis zu einer dynamisch zu ermittelnden Stelle kopiert werden und vor das zu bearbeitende Programm gehängt werden (siehe Tabelle zur Funktion g auf Seite 14).

Der neue Interpreter arbeitet mit zwei Variablen vom Typ CRegister. Eine Variable enthält dabei den Programmcode (Stapel) und die andere Variable den aktuellen Inhalt der Register (Tupel). Zur Konvertierung des Register-Maschinen-Programms¹⁶ werden folgende Regeln verwendet:

¹⁶ Der Grund für diese Codierung ist in der Bestrebung nach einfacher Verarbeitung für die Register-Maschine und Lesbarkeit für den Menschen zu finden.

1. Die Registerwerte werden in unäre Zahlen gewandelt und durch Nullen getrennt aneinandergelängt. Wobei dann aus 1,2,0,4 die Zahl 1011001111 wird.
2. Der Befehlscode wird nach folgenden Regeln umgewandelt:

Befehlscode	Umwandlung	Beispiel Code	Beispiel Umwandlung
A _i	2<unär i>	A2	<u>211</u>
S _i	3<unär i>	S2	<u>311</u>
(5	(S2)3	<u>53114111</u>
) _i	4<unär i>		

Tabelle II-3: Umwandlungstabelle des Befehlscodes für den Interpreter

Im nächsten Schritt werden die erzeugten Zahlen dann gespiegelt, sodass die höchstwertige Ziffer am Ende der Ziffernkette liegt. Dies wird durchgeführt, um die weitere Verarbeitung zu vereinfachen: das Auslesen des nächsten „Codes“ des Programms wird durch die Operation Modulo 10 erreicht, das Entfernen von Codes durch die Division mit 10. Die so erhaltenen Zahlen bilden die Ausgangsbasis für die Funktion g.

Beispiel: (A3 A4 S1)1 (A1 S4)4 (A3 A4 S2)2 (A2 S4)4 . 5, 7, 0
wird umgewandelt in Befehlscode

111141111311251141131111211125111141111312514131111211125
und Registerinhalt 1111111011111

(Um die Lesbarkeit zu erhöhen, habe ich die zusammengehörigen Ziffern gleich unterstrichen.)

Der Interpreter betrachtet nun im ersten Schritt die letzte Ziffer, indem er den Registerinhalt des Programmregisters Modulo 10 berechnet. (Im obigen Beispiel wäre dies die 5.) Dann wird geprüft, ob dies Addition, Subtraktion oder Schleifenbeginn ist und die entsprechende Unterfunktion aufgerufen. Dies ist das generelle Vorgehen nach Funktion g.

Addition und Subtraktion sind ähnlich implementiert. Zur Verdeutlichung habe ich die Addition als detailliertes Struktogramm angegeben, dazu ein Beispiel der Inhalte der Variablen für Register-Maschinen-Code und Register-Maschinen-Status.

Addition

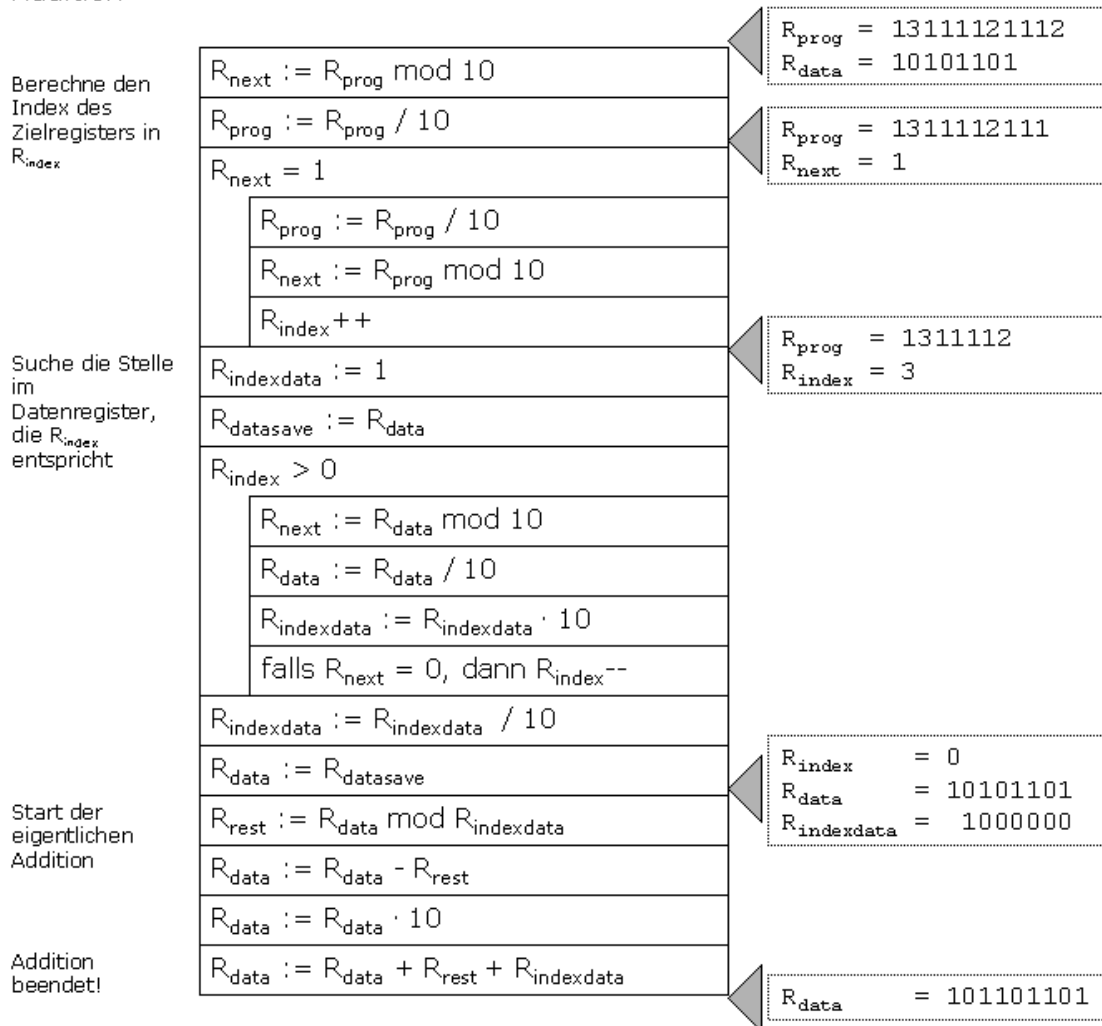


Abbildung II-6: Struktogramm des Additions-Teils des Interpreters

Die Bearbeitung der Subtraktion entspricht in etwa der Addition, die der Schleife ist komplexer: Zunächst muss das Ende der Schleife gefunden werden, dies geschieht mit einem Zähler, der die Schleifentiefe angibt. Danach muss zunächst getestet werden, ob die Schleife durchlaufen werden soll, dazu muss der Wert des zur Schleife gehörenden Registers ausgelesen werden. Ist das Schleifen-Register nicht Null, so wird der von der Schleife umschlossene Programmcode mit einer Kombination von Modulo-Operationen und Multiplikationen kopiert und (wegen der gespiegelten Kodierung) hinter der Schleife eingefügt. Ist das Schleifen-Register Null, so kann die gesamte Schleife mit einer einzelnen Division entfernt werden.

III. Der Compiler

Im Kapitel II wurde die Register-Maschine vorgestellt, deren Funktionsweise durch eine Compiler-Semantik (siehe Seite 12) genau definiert ist; damit war ein Übersetzer von Register-Maschinen-Code in C++-Code möglich. Die folgenden Kapitel beschäftigen sich mit dem umgekehrten Weg: es wird versucht, ausgehend von der Compiler-Semantik, Übersetzungsregeln von C++ nach Register-Maschinen-Code zu finden. Dazu wird zunächst ein allgemeiner Parser für eine C++-Syntax erstellt und später in diesen Parser an geeigneten Stellen der Übersetzer eingeflochten.

Für den Compiler wurden verschiedene Klassen, Strukturen, Methoden und Variablen erstellt, die an dieser Stelle nur aufgelistet werden, in den Unterkapiteln wird dann auf die einzelnen Bestandteile genauer eingegangen. (Die vollständigen Tabellen befinden sich im Anhang auf Seite 83)

CErrorCompiler	
Dient dem Compiler als Abbruchkriterium. Wird in der Klasse CParse in den verschiedenen Parse-Funktionen verwendet um einen Fehler anzuzeigen, der nicht behoben werden kann, und der zum Stopp des Compilierens führen soll.	
Methode/Eigenschaft	Beschreibung
CErrorCompiler	Konstruktor
~CErrorCompiler	Destruktor
Cause	liefert den Grund für den Abbruch. Der Grund ist z. Zt. fest auf "ERROR" gestellt.

Tabelle III-1: Klasse CErrorCompiler

CToken	
Liest von einem Eingabestrom (Datei) und kann über einen Puffer Token zurückliefern. Stellt eine Funktion zur Ausgabe in einen Ausgabestrom (Datei) und eine Logging-Funktion zur Verfügung. Wird von CParse benutzt.	
Methode/Eigenschaft	Beschreibung
CToken	Konstruktor
~CToken	Destruktor
NextToken	liefert das nächste Token
PutBack	schiebt die gegebene Zeichenkette zurück in den Puffer, diese Zeichenkette wird als nächstes aus dem Puffer zurückgeliefert
Write	schreibt in die Ausgabedatei

Tabelle III-2: Klasse CToken

C_Parse	
Parser für C++-Code. Enthält zusätzlich zur reinen parse-Funktionalität noch den Code des Übersetzers.	
Methode/Eigenschaft	Beschreibung
C_Parse	Konstruktor
~C_Parse	Destruktor
SObject * psTop;	Zeiger auf die doppelt verkettete Liste der Variablen und Funktionen
AddObject	fügt Variable, Funktion oder Wert in die SObject-Liste an der richtigen Stelle hinzu
FindObject	sucht einen Eintrag in der SObject-Liste
AddCode	fügt Code an die Funktion in der SObject-Liste an
ParseProgram	compiliert die im Konstruktor angegebene Datei
ParseFunction	bearbeitet Funktionsdefinitionen
ParseFunctionType	bearbeitet den Funktionstyp der Funktionsdefinition
ParseNewFunctionName	bearbeitet den Namen der Funktionsdefinition: trägt den neuen Namen in die SObject-Liste ein
ParseParameterList	bearbeitet die (ev. leere) Parameterliste in der Funktionsdefinition
ParseFilledParameterList	bearbeitet eine nicht leere Parameterliste in der Funktionsdefinition
ParseDeclaration	bearbeitet eine einzelne Deklaration einer Übergabevariablen oder einer Deklaration einer lokalen Variablen
ParseInitValue	bearbeitet die Zuweisung eines Initialwertes
ParseVariableType	bearbeitet den Typ der Variable bei einer Deklaration
ParseNewVariable	bearbeitet den Namen einer Variable bei einer Deklaration: fügt den neuen Namen in die SObject-Liste ein
ParseBlock	bearbeitet den Funktionsrumpf
ParseDeclarationList	bearbeitet eine (ev. leere) Deklarationsliste für lokale Variablen innerhalb einer Funktion
ParseTermList	bearbeitet eine (ev. leere) Anweisungsliste innerhalb einer Funktion
ParseTerm	bearbeitet eine einzelne Anweisung (if/if-else/while/Zuweisung)
ParseIfTerm	bearbeitet eine If-Anweisung
ParseWhileTerm	bearbeitet eine While-Schleife
ParseCondition	bearbeitet die Bedingung in einer if-Anweisung oder einer While-Schleife
ParseNotExpression	bearbeitet die Negation eines Ausdrucks
ParseNotOperator	bearbeitet den Negationsoperator

C_Parse	
Parser für C++-Code. Enthält zusätzlich zur reinen parse-Funktionalität noch den Code des Übersetzers.	
Methode/Eigenschaft	Beschreibung
ParseLogicalExpression	bearbeitet logische Ausdrücke
ParseKnownVariableNumber	bearbeitet bekannte Variablen oder Zahlen-Konstanten
ParseKnownVariable	bearbeitet bekannte Variablen
ParseAssignment	bearbeitet Zuweisungen
ParseFunctionCall	bearbeitet Funktionsaufrufe (durch Call-By-Copy implementiert)
ParseKnownFunctionName	bearbeitet bekannte Funktionsnamen
ParseCallParameterList	bearbeitet die (ev. leere) Übergabeliste der Parameter eines Funktionsaufrufs
ParseFilledCallParameterList	bearbeitet eine nichtleere Übergabeliste der Parameter eines Funktionsaufrufs
ParseCallParameter	bearbeitet einen einzelnen Übergabeparameter
ParsePostfixOperator	bearbeitet die Postfix-Operatoren (++) und (--)
ParseAssignmentOperator	bearbeitet die Zuweisungsoperatoren =, +=, -=, *=, /=, %=
ParseOperator	bearbeitet einen arithmetischen Operator +, -, *, /, %
ParseLogicalOperator	bearbeitet logische Operatoren (momentan nur == und !=)
ParseRelationOperator	bearbeitet relationale Operatoren (>, >=, <, <=)
ParseNumber	bearbeitet Zahlen-Konstanten
ParseExpected	bearbeitet erwartete Eingabezeichen
ParseSemicolon	bearbeitet „;“
ParseBlockStart	bearbeitet „{“
ParseBlockEnd	bearbeitet „}“
ParseListStart	bearbeitet „(“
ParseListEnd	bearbeitet „)“
ParseListSeparator	bearbeitet „,“ (Komma)

Tabelle III-3: Klasse C_Parse

SObject	
Nimmt Informationen zu Variablen und Funktionen auf. Erzeugt eine mehrfach verkettete Liste.	
Methode/Eigenschaft	Beschreibung
char * pcName	Name der Variable oder Funktion
char * pcCode	Code zur Funktion
eOBJECT_TYPE eType	Typ des Listenelements
SObject * psNext	nächstes Listenelement
SObject * psLast	vorhergehendes Listenelement

SObject	
Nimmt Informationen zu Variablen und Funktionen auf. Erzeugt eine mehrfach verkettete Liste.	
Methode/Eigenschaft	Beschreibung
SObject * psDown	untergeordnetes Listenelement
SObject * psUp	übergeordnetes Listenelement

Tabelle III-4: Struktur SObject

1. Generelle Methodik

Zur Übersetzung der Programme wurde eine Klasse CToken erstellt, die aus einem Eingabestrom (Textfile) jeweils ein für die gegebene C++-Syntax gültiges Token liefert. Diese Klasse kümmert sich selbst um interne Lesepuffer und erlaubt es auch Token wieder in den Puffer zurückzugeben. Diese Klasse stellt eine Art Scanner dar. Die Klasse CToken wird von der Klasse CParse verwendet - diese implementiert den eigentlichen Compiler. Die Vorgehensweise des Compilers richtet sich streng nach der EBNF-Darstellung der verwendeten C++-Syntax. Dabei existiert für jeden Bezeichner eine Funktion, die prüft, ob der Bezeichner zum nächsten Token im Eingabestrom passt (dies sind die „Test...“-Funktionen) und die danach die nächsten Funktionen für die Folgebezeichner aufruft (rekursiver Abstieg).

Beispiel `if (a == b) { c = 1; }`: die Funktion `CParse::ParseTerm()` nimmt jeden Ausdruck innerhalb eines Blocks entgegen, und ruft, nachdem durch `if` der `if`-Ausdruck erkannt wurde, die Funktion `CParse::ParseIfTerm()` auf. Diese Funktion wiederum ruft `CParse::ParseCondition()`, `CParse::ParseBlock()` und, falls ein `else` zum `if` gehört, noch ein zweites `CParse::ParseBlock()` auf.

Der während des Compilierens erzeugte Code wird jeweils in einem Speicherbereich abgelegt, der durch die gerade zu compilierende Funktion identifiziert wird. Am Ende des Compilierens wird dann der Register-Maschinen-Code für die Funktion `main()` in eine Datei gesichert. Dies ist möglich, weil die Funktionsaufrufe durch Ersetzung des Aufrufs mit dem eigentlichen Code realisiert ist. Das damit entstandene Verbot von rekursiven Funktionen ist in der verwendeten C++-Syntax impliziert, da keine Deklarationen von Funktionen ohne deren Definition erlaubt sind.

Auf die Verwendung von Lex und Yacc wurde verzichtet, da die benötigte C++-Syntax ausreicht, um den Interpreter für Register-Maschinen-Programme zu erstellen. In einem nächsten Schritt könnte der Interpreter verfeinert werden und Konstanten, etc. enthalten. Dann wäre der Einsatz von Lex und Yacc sicherlich sinnvoll.

2. Verwendete C++ Syntax

Die hier verwendete C++-Syntax ist stark an ANSI C++¹⁷ angelehnt. Es gibt nach der verwendeten Definition keine globalen Variablen. Dadurch, dass momentan kein Präprozessor verwendet wird, sind auch keine Präprozessor-Ausdrücke möglich (`#define`, `#include`, etc.).

¹⁷ Stroustrup, B.: The C++ Programming Language.

Bezeichner	Ausdruck	Anmerkung
program	""	
	function program	
function	function-type new-function-name "(" parameter-list ")" block	
function-type	"void"	nur „void“ ist gültiger Typ für Funktionen
new-function-name	name	unbekannter Funktionsname
parameter-list	""	
	filled-parameter-list	
filled-parameter-list	declaration	
	declaration "," filled-parameter-list	
declaration	variable -type new-variable	
	variable -type new-variable "=" number	
variable-type	"CRegister"	nur "CRegister" ist gültiger Typ für Variablen
new-variable	name	(lokal) unbekannter lokaler Variablenname
block	"{" declaration-list term-list "}"	
declaration-list	""	
	declaration ";" declaration-list	
term-list	""	
	term ";" term-list	
term	if-term	
	while-term	
	assignment	
if-term	"if" condition block	
	"if" condition block "else" block	
while-term	"while" condition block	
condition	(" logical-expression ")	
logical-expression	"!" condition	

Bezeichner	Ausdruck	Anmerkung
	condition logical-operator condition	
	known-variable-number relation-operator known-variable-number	
known-variable-number	known-variable	bekannte (deklarierte) Variable oder Zahlenkonstante
	number	
known-variable	name	bekannte (deklarierte) Variable
assignment	known-variable postfix-operator	
	known-variable assignment-operator known-variable-number	nur „a=b“ oder "a=b+c" erlaubt!
	known-variable set-operator known-variable-number operator known-variable-number	
	function-call	
function-call	known-function "(" call-parameter-list ")"	bekannte (definierte) Funktion
known-function-name	name	
call-parameter-list	""	
	filled-call-parameter-list	
filled-call-parameter-list	call-parameter	
	call-parameter "," filled-call-parameter-list	
call-parameter	known-variable-number	
postfix-operator	"--" "++"	
set-operator	"="	
assignment-operator	set-operator "+=" "-=" "*=" "/=" "%="	
operator	"+" "-" "*" "/" "%"	
logical-operator	"==" "&&" " "	
relation-operator	"==" "<" "<=" ">" ">=" "!="	

Tabelle III-5: EBNF-Darstellung der verwendeten C++ Syntax

3. Übersetzung der C++ Syntax

Bevor ich auf die Übersetzung selbst zu sprechen komme, möchte ich noch auf ein paar Besonderheiten hinweisen: Ich habe mich entschlossen keine negativen Zahlen zuzulassen. Jede Operation, die Zahlen kleiner als Null liefern würde, liefert die Zahl Null zurück.

Um alle Funktionalitäten implementieren zu können, war es notwendig einige Hilfsregister zu definieren.

Symbolischer Registername	Index	Beschreibung
R _{cond}	9	Ergebnis einer Auswertung. Wird bei „if“ und „while“ verwendet
R _{else}	1	Ergebnis einer „else“-Auswertung. Wird bei „if“ verwendet
R _{copy}	2	wird bei verschiedenen Kopiervorgängen (Zuweisungen, etc.) verwendet
R _{h1}	3	wird bei Auswertungen als Hilfsregister benötigt
R _{h2}	4	wird bei Auswertungen als Hilfsregister benötigt
R _{while}	5	wird bei der „while“-Schleife verwendet
R _{mod}	6	wird bei der Modulo Operation verwendet
R _{value}	7	wird beim Übergang von modifizierenden Zuweisungen zu Rechenoperationen verwendet

Tabelle III-6: Festlegung der Hilfsregister

3.1. Variablen und Funktionen

Jede Variable (und in der verwendeten C++-Syntax gibt es nur lokale Variablen) wird durch ein eigenes (globales) Register repräsentiert. Dadurch muss eine genügend dimensionierte Register-Maschine als Zielplattform dienen; da aber der vorgestellte Interpreter eine nur durch den Arbeitsspeicher begrenzte Anzahl an Registern simulieren kann, stellt dies kein Problem dar. Es wird sich auch zeigen, dass mit der vorgestellten Methode nur etwa 50 Register benötigt werden.

Findet der Compiler die Deklaration einer neuen Variable, wird diese in einer doppelt verketteten Liste vom Typ SObject abgelegt, und zwar unter dem Eintrag der zugehörigen Funktion (s.u.). In der Struktur befindet sich der Name, der Initialwert (falls vorhanden), das zugehörige Register, die Angabe über die Funktion, in der die Variable deklariert wurde und Zeiger auf die vorhergehende und nächste Variable.

Der bei der Übersetzung von Funktionen entstehende Code wird ebenfalls in der mehrfach verketteten Liste der SObject-Variable gehalten. Die „obersten“ Knoten der Liste enthalten Informationen zu den Funktionen, Name, Code, vorhergehende und nachfolgende Funktion und ein Zeiger auf die zugehörigen Variablen in der Funktion.

Funktionen werden durch Einfügen des übersetzten Codes „aufgerufen“; aus diesem Grund müssen Funktionen vor ihrer Verwendung definiert werden. Rekursive Funktionsaufrufe können also nicht übersetzt werden - dies stellt allerdings keine echte Einschränkung in der Funktionalität dar - rekursive Funktionsaufrufe können zu Schleifen umformuliert werden. Auf die Problematik der globalen Variablen und des Verbots von rekursiven Funktionen werde ich im Ausblick zum Übersetzer näher eingehen (siehe Seite 42).

Die Variable vom Typ „SObject“, die alle Daten über Funktionen und Variablen aufnimmt, ist eine Variable der Klasse CParse, von der ein einziges Objekt zur Übersetzungszeit erstellt wird.

Name & Typ
Code oder Index & Initialwert
Zeiger auf nächstes und vorhergehendes Objekt
Zeiger auf zugehörige Variable oder Funktion

Abbildung III-1: Struktur SObject

Zur Verdeutlichung der Verwendung der SObject-Liste, hier zunächst noch ein kleines Beispiel. Dabei gibt es zwei Funktionen „square“ und „main“; „square“ hat lokale Variablen oder Parameter „a“ und „b“, „main“ hat lokale Variable oder Parameter „sqr“:

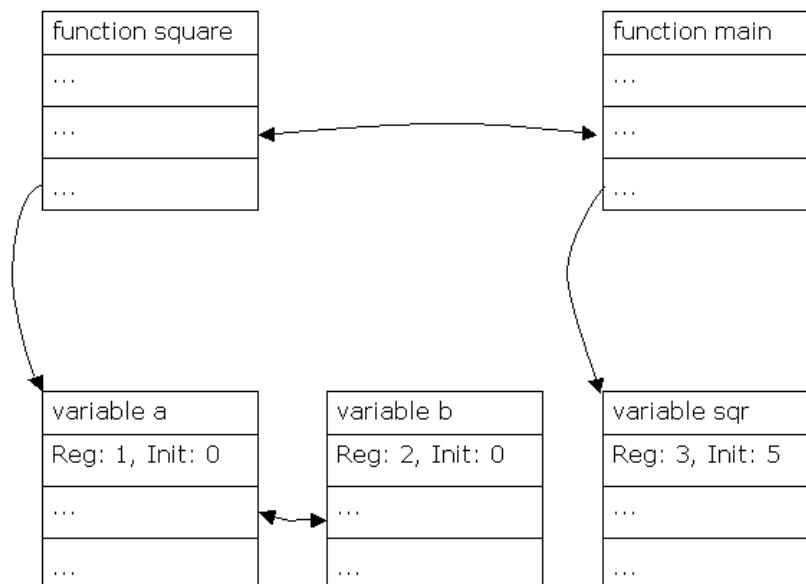


Abbildung III-2: Beispiel für die Verwendung der Struktur SObject

3.2. Übersetzung von Zuweisungen und Rechenoperationen

Es gibt verschiedene Arten von Zuweisungen und Rechenoperationen, die jeweils unterschiedlich übersetzt werden müssen. Zunächst werden die Zuweisungen und Rechenoperationen vorgestellt und eingeteilt, um dann später im Einzelnen behandelt zu werden.

1. Operationen, die direkt auf eine Variable wirken
 $v++$ oder $v--$
2. Zuweisungen von Werten oder Variablen auf Variablen
 $v = 1$ oder $v = w$
3. modifizierende Zuweisungen von Werten oder Variablen auf Variablen
 $v += 1$ (addiert eins zu v) oder $v -= w$ (subtrahiert w von v)
4. Zuweisungen von Ergebnissen einfacher Rechenoperationen auf Variablen
 $v = 1 + 3$ oder $v = w - 2$

3.2.1 Übersetzung von „++“ und „--“

Die Operationen „++“ und „--“ sind direkt zu übersetzen: aus „ $v++$ “ wird „ A_i “ wobei i der Index des Registers zur Variable v ist.

Bei der Übersetzung von „--“ habe ich die einfache, aber unkorrekte, Möglichkeit „ $v--$ “ wird zu „ S_i “ gewählt. Diese Übersetzung ist deshalb nicht ganz korrekt, da „ $v--$ “ angewendet auf Null eine negative Zahl liefern sollte.

3.2.2 Übersetzung von einfachen Zuweisungen

Bei einer Zuweisung „ $=$ “ bei der einer Variablen eine Zahl zugewiesen werden soll, wird das der Variablen entsprechende Register mit $(S_i)_i$ initialisiert und dann mit entsprechend vielen A_i auf den Wert der Zahl erhöht.

Die Übersetzung der einfachen Zuweisung $R_{links} = R_{rechts}$ wird unter Verwendung des Hilfsregisters R_{copy} durch $(S_{links})_{links} (S_{copy})_{copy} (A_{links} A_{copy} S_{rechts})_{rechts} (A_{rechts} S_{copy})_{copy}$ realisiert.

3.2.3 Übersetzung von modifizierenden einfachen Zuweisungen

- Die Addition „ $+=$ “ entspricht der Zuweisungsoperation ohne vorheriges Löschen des Registers. Das Zielregister wird mittels einer Schleife so oft erhöht und das Quellregister so oft erniedrigt, bis das Quellregister den Wert Null erreicht. Gleichzeitig wird das Hilfsregister R_{copy} zum Sichern des alten Inhaltes des

Quellregisters verwendet.

$$R_{links} += R_{rechts} \rightarrow (S_{copy})_{copy} (A_{links} A_{copy} S_{rechts})_{rechts} (A_{rechts} S_{copy})_{copy}$$

- Im Unterschied zu „+=“ wird bei Subtraktion „-=“ in der Schleife statt A der Operator S verwendet.

$$R_{links} -= R_{rechts} \rightarrow (S_{copy})_{copy} (S_{links} A_{copy} S_{rechts})_{rechts} (A_{rechts} S_{copy})_{copy}$$

- Die Multiplikation funktioniert nach dem Schema der wiederholten Addition. Dazu wird zunächst der ursprüngliche (und mehrfach zu addierende) Wert der linken Seite in ein Hilfsregister kopiert und die linke Seite auf Null gesetzt. Dann wird eine Schleife über das Register der rechten Seite durchlaufen, dieses jeweils um Eins erniedrigt und der zu addierende Wert auf das Register der linken Seite addiert.

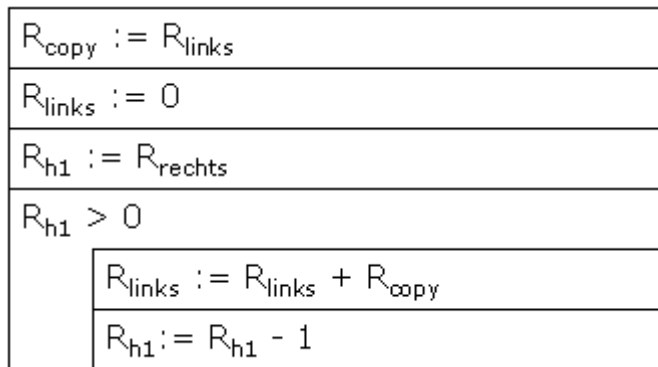


Abbildung III-3: Struktogramm der Multiplikation „*=“

$$R_{links} *= R_{rechts} \rightarrow (S_{copy})_{copy} (S_{h1})_{h1} (S_{h2})_{h2} (A_{copy} S_{links})_{links} (A_{h1} A_{h2} S_{rechts})_{rechts} (S_{h2} A_{rechts})_{h2} (S_{h1} (S_{copy} A_{links} A_{h2})_{copy} (S_{h2} A_{copy})_{h2})_{h1} (S_{copy})_{copy}$$

- Die Division funktioniert nach dem Schema der wiederholten Subtraktion. Dazu wird zunächst der ursprüngliche (und mehrfach zu subtrahierende) Wert der linken Seite in ein Hilfsregister kopiert. Dann wird eine Schleife über das Hilfsregister mit dem ursprünglichen Wert der linken Seite durchlaufen: von diesem Hilfsregister wird der Wert der rechten Seite abgezogen und zur linken Seite Eins addiert. Nach Beendigung der Schleife enthält das Register der linken Seite die Anzahl der Schleifendurchläufe und damit den Wert der Divisionsoperation. Dieses Verfahren funktioniert nicht richtig bei Divisionen, die einen Rest haben (also etwa „30 dividiert durch 8“, dies hat Rest 6), dort liefert die beschriebene Methode einen um Eins höheres Ergebnis: Die Schleife wird nochmals durchlaufen, obwohl nicht alle Subtraktionen den Wert des Registers verkleinern können (da der Wert schon Null ist!). Beispiel: $R_{links} = 5$ und $R_{rechts} = 8$. Die Schleife wird mindestens einmal durchlaufen, da die Schleife über R_{copy} läuft und dieses beim Schleifenstart 7 enthält. Dieses Problem kann man lösen, indem man vor Beginn der

Schleife den Wert „Divisor – 1“ vom Divident abzieht.

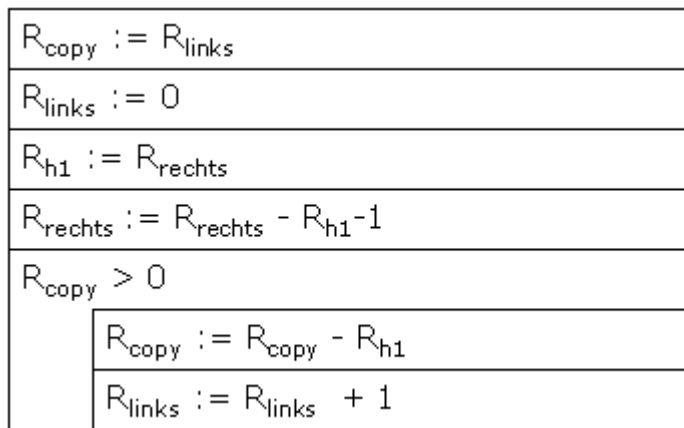


Abbildung III-4: Struktogramm der Division „/“

$R_{\text{links}} / = R_y \rightarrow (S_{\text{copy}})_{\text{copy}} (S_{\text{h1}})_{\text{h1}} (S_{\text{h2}})_{\text{h2}} (A_{\text{copy}} S_{\text{links}})_{\text{links}} (A_{\text{h1}} A_{\text{h2}} S_{\text{rechts}})_{\text{rechts}} (S_{\text{h2}} A_{\text{rechts}})_{\text{h2}} S_{\text{h1}} (S_{\text{copy}} S_{\text{h1}} A_{\text{h2}})_{\text{h1}} (A_{\text{h1}} S_{\text{h2}})_{\text{h2}} A_{\text{h1}} ((S_{\text{copy}} S_{\text{h1}} A_{\text{h2}})_{\text{h1}} (A_{\text{h1}} S_{\text{h2}})_{\text{h2}} A_{\text{links}})_{\text{copy}} (S_{\text{h1}})_{\text{h1}}$

- Die Modulo Operation ist eine Kombination von Multiplikation, Division und Subtraktion. Zunächst wird eine Division mit dem Wert des Registers der linken Seite durchgeführt und das Ergebnis dann wieder mit der rechten Seite multipliziert. Die Differenz der nun erhaltenen Zahl (abgelegt im Register R_{mod}) und des ursprünglichen Wertes des Registers der linken Seite ist das Ergebnis der Modulo Operation.

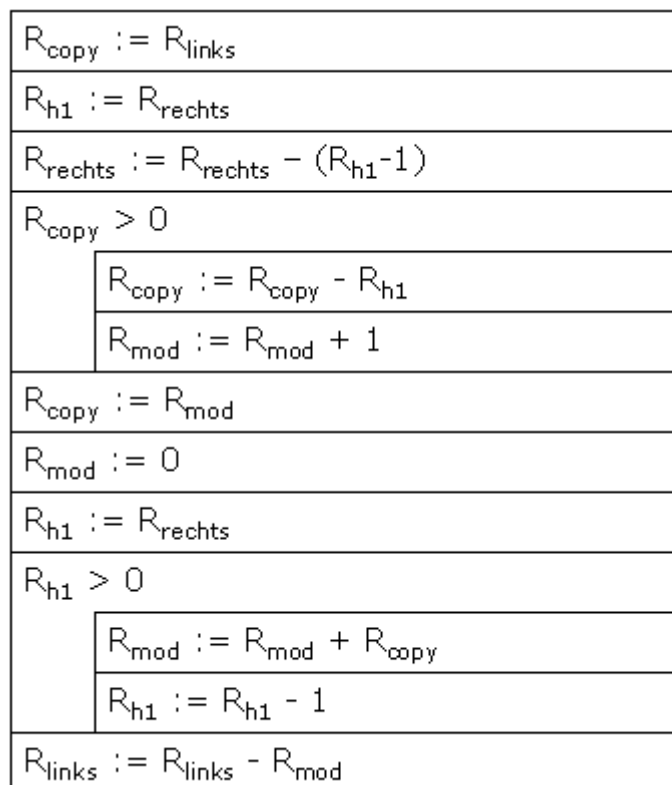


Abbildung III-5: Struktogramm der Modulo Operation „%“

$$\begin{aligned}
R_{links} \% = R_{rechts} &\rightarrow (S_{copy})_{copy} (S_{h1})_{h1} (S_{h2})_{h2} (S_{mod})_{mod} (A_{copy} A_{h1} \\
S_{links})_{links} (S_{h1} A_{links})_{h1} (A_{h1} A_{h2} S_{rechts})_{rechts} (S_{h2} A_{rechts})_{h2} S_{h1} \\
(S_{copy} S_{h1} A_{h2})_{h1} (A_{h1} S_{h2})_{h2} A_{h1} ((S_{copy} S_{h1} A_{h2})_{h1} (A_{h1} S_{h2})_{h2} \\
A_{mod})_{copy} (S_{h1})_{h1} (A_{copy} A_{h1} S_{mod})_{mod} (S_{h1} A_{mod})_{h1} (A_{h1} A_{h2} \\
S_{rechts})_{rechts} (S_{h2} A_{rechts})_{h2} S_{h1} (S_{h1} (S_{copy} A_{mod} A_{h2})_{copy} (S_{h2} \\
A_{copy})_{h2})_{h1} (S_{copy})_{copy} (S_{links} S_{mod})_{mod}
\end{aligned}$$

Alle fehlenden Rechenoperationen sind für den zu übersetzenden Interpreter nicht notwendig, können aber auf die gegebenen Rechenoperationen zurückgeführt werden.

3.2.4 Übersetzung von Rechenoperationen

Die Übersetzung aller Rechenoperationen, also Zuweisungen des Typs „ $V_1 = V_2 \text{ op } V_3$ “ sind nicht implementiert. Sie können aber leicht durch modifizierende Zuweisungen ersetzt werden. Im Fall von „ $V_1 = V_2 \text{ op } V_3$ “ kann dies durch „ $V_1 = V_2; V_1 \text{ op} = V_3$ “ geschehen. Verschachtelte Rechenoperationen müssten ebenso umformuliert werden. Dies könnte die Aufgabe eines Präprozessors oder Lex-Tools sein.

3.3 Übersetzung von „if“ / „while“ und Bedingungen

Die Übersetzung der beiden Konstrukte „if“ und „while“ kann unterteilt werden in die Übersetzung eines Rahmens und die Übersetzung der Bedingung. Zunächst werde ich auf die Übersetzung der Bedingung eingehen und dann die beiden unterschiedlichen Rahmen besprechen.

3.3.1 Übersetzung von Bedingungen

Alle Bedingungen werden als Berechnungen übersetzt. Das Ergebnis dieser Berechnungen wird in einem festen Register (R_{cond}) abgelegt. Eine wahre Bedingung soll dabei Wert ungleich Null im Register R_{cond} zur Folge haben, eine unwahre Bedingung den Wert Null.

- Der einfachste Test ist hierbei der Test auf ungleich Null. Dabei wird der Inhalt des zu testenden Registers in das Bedingungsregister kopiert.

$$R_{links} \quad 0 \rightarrow (S_{links} A_{copy} A_{cond})_{links} (S_{copy} A_{links})_{copy}$$

- Der Test auf Gleichheit wird über die Berechnung der Differenz von R_{links} und R_{rechts} ausgeführt. Es wird $| R_{links} - R_{rechts} | + | R_{rechts} - R_{links} |$ berechnet. (Zu beachten ist dabei, dass die beiden Differenzen mit Register-Maschinen berechnet nicht den gleichen Wert liefern, da die Subtraktion dabei nicht vertauschbar ist! Ist R_{links} größer als R_{rechts} , so liefert $R_{links} - R_{rechts}$ genau Null und $R_{rechts} - R_{links}$ einen positiven Wert.) Die Summe der Differenzen entspricht der Berechnung der Ungleichheit und ist Null, falls beide gleich sind, ansonsten

ungleich Null. Dieser Wert wird dann invertiert, d.h. Null wird zu Eins und jeder Wert ungleich Null wird zu Null.

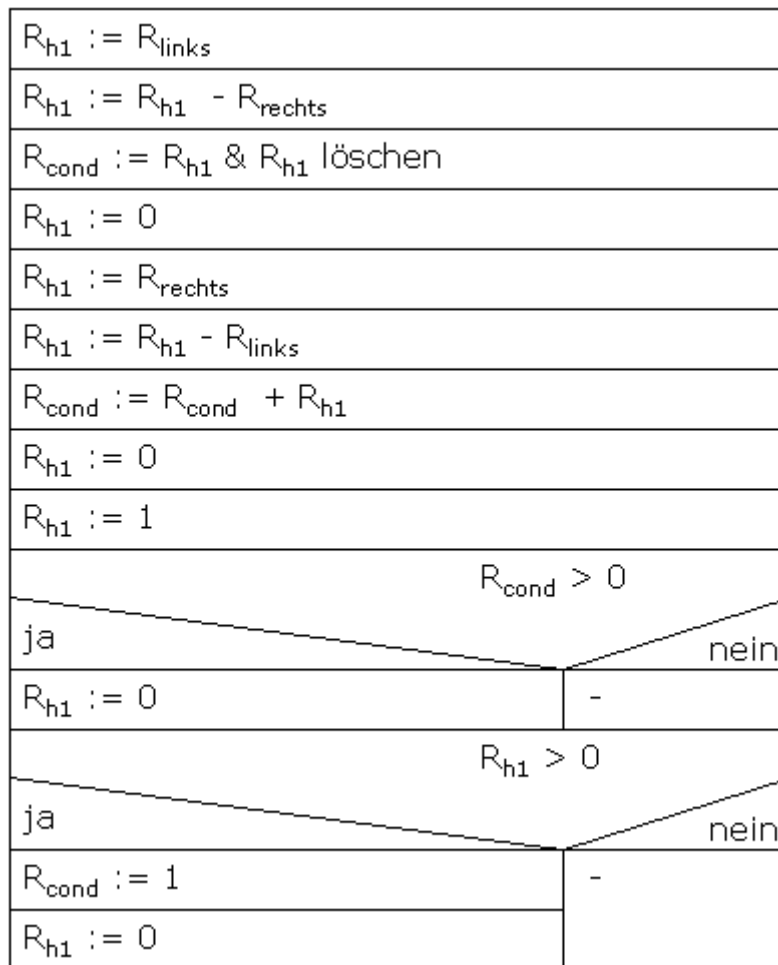


Abbildung III-6: Struktogramm des Tests auf Gleichheit

$R_{links} = R_{rechts} \rightarrow (S_{cond})_{cond} (S_{h1})_{h1} (S_{h2})_{h2} (S_{links} A_{h1} A_{h2})_{links}$
 $(A_{links} S_{h2})_{h2} (S_{rechts} S_{h1} A_{h2})_{rechts} (A_{rechts} S_{h2})_{h2} (A_{cond} S_{h1})_{h1}$
 $(S_{rechts} A_{h1} A_{h2})_{rechts} (A_{rechts} S_{h2})_{h2} (S_{links} S_{h1} A_{h2})_{links} (A_{links}$
 $S_{h2})_{h2} (A_{cond} S_{h1})_{h1} A_{h1} (S_{cond} S_{h1})_{cond} ((S_{h1})_{h1} A_{cond})_{h1}$

- Der Test auf Ungleichheit entspricht dem Test auf Gleichheit ohne die anschließende Invertierung.

$R_{links} \neq R_{rechts} \rightarrow (S_{cond})_{cond} (S_{h1})_{h1} (S_{h2})_{h2} (S_{links} A_{h1} A_{h2})_{links}$
 $(A_{links} S_{h2})_{h2} (S_{rechts} S_{h1} A_{h2})_{rechts} (A_{rechts} S_{h2})_{h2} (A_{cond} S_{h1})_{h1}$
 $(S_{rechts} A_{h1} A_{h2})_{rechts} (A_{rechts} S_{h2})_{h2} (S_{links} S_{h1} A_{h2})_{links} (A_{links}$
 $S_{h2})_{h2} (A_{cond} S_{h1})_{h1}$

Die hier nicht angegebenen, aber in der Definition der C++-Syntax enthaltenen Vergleiche, sowie die logischen Operationen haben sich für den gegebenen Interpreter als unnötig erwiesen und wurden deshalb nicht in den Compiler implementiert.

3.3.2 Übersetzung von „if“

Die Schwierigkeit bei der Übersetzung des "if" Ausdrucks liegt darin, dass zum einen der enthaltene if-Block nur einmal ausgeführt werden soll und dass zum anderen der else-Block durchlaufen werden kann. Zusätzlich sollen auch geschachtelte "if" Ausdrücke korrekt übersetzt werden können. Nachdem die Bedingung berechnet wurde, wird der Ausdruck

- $if\ condition\ \{block\ 1\}\ else\ \{block\ 2\} \rightarrow compute\ condition\ (S_{else})_{else}\ A_{else}(block\ 1\ (S_{else})_{else}\ (S_{cond})_{cond}\)_{cond}\ (block\ 2\ (S_{else})_{else}\)_{else}$

übersetzt.

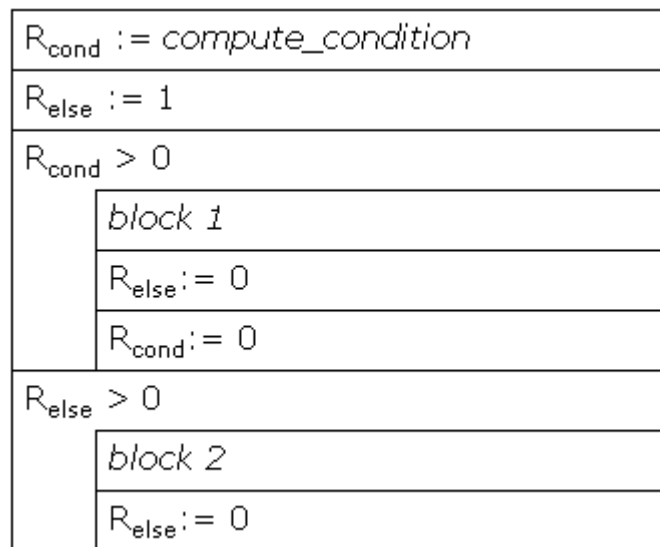


Abbildung III-7: Übersetzung der if- Anweisung

Durch das Löschen der Schleifenregister in der Schleife selbst wird sichergestellt, dass jede Schleife nur jeweils einmal durchlaufen wird. Zusätzlich stellt das Löschen der Schleifenregister noch sicher, dass bei geschachtelten "if" Ausdrücken keine Interferenz zwischen den Bedingungsregistern entstehen.

3.3.3 Übersetzung von „while“

Die while-Schleife hat etwas Probleme bereitet, da dort die Eigenarten der Register-Maschine besonders deutlich werden. Der erste Versuch, eine Abwandlung der if-Anweisung, war erfolglos: die Schwierigkeit besteht darin, die Bedingung an die richtige Stelle zu setzen, damit sie mindestens einmal, aber auch n-mal aufgerufen werden kann. Es hilft, dass beim umgekehrten Weg die Register-Maschinen-Schleife direkt mit einer While-Schleife übersetzt und dass bei der Abarbeitung durch die Funktion g (Seite 14) der Schleifen-Inhalt vor die Schleife kopiert wird. Daraus ergibt sich, die recht einfache Übersetzung zu:

- $while\ condition\ \{block\} \rightarrow A_{while}\ A_{cond}\ ((S_{while})_{while}\ compute\ condition\ (block\ A_{while}\ (S_{cond})_{cond}\)_{cond}\)_{while}$

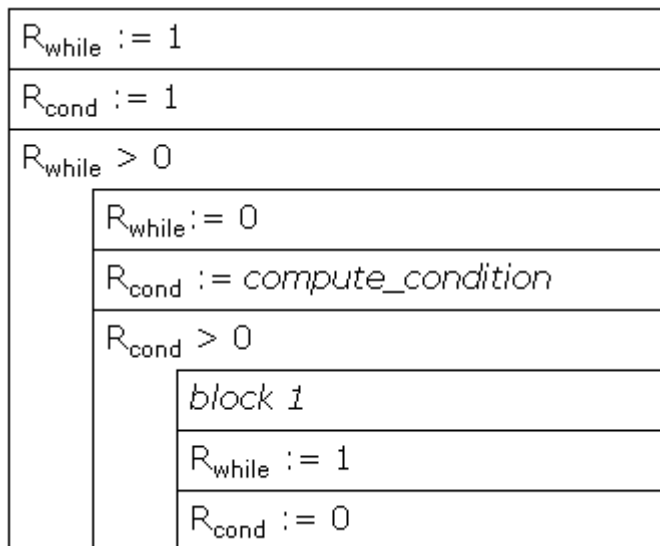


Abbildung III-8: Übersetzung der while-Anweisung

4. Übersetzer – Ausblick

Um den Interpreter für Register-Maschinen-Programme zu übersetzen, reicht der vorgestellte Compiler aus. Allerdings sind weder Kommentare noch kompliziertere Zuweisungen, Konstanten oder geschachtelte Bedingungen erlaubt. Dies ist der Übersichtlichkeit nicht gerade dienlich.

Die nächste Stufe wäre es, den Compiler mit Lex und Yacc zusammenzuführen. Dann könnten Makros, Präprozessor-Definitionen und das Einbinden von anderen Quelltexten möglich sein. Geschachtelte Bedingungen und komplexere Berechnungen könnten durch Lex in einfachere Anweisungen überführt werden und der Compiler selbst bräuchte nicht weiter ausgebaut zu werden.

Ein größeres Problem stellt allerdings die Verwendung von rekursiven Funktionen dar. Der Grund, dass dies nicht möglich ist, liegt zum einen daran, dass im vorgestellten Compiler alle Funktionsaufrufe durch die jeweilige Funktion selbst ersetzt werden und dass zum anderen nur globale Variablen benutzt werden.

Im Register-Maschinen-Code von Ottmann und Albert gibt es keine Funktionsaufrufe und keine lokalen Variablen (siehe Seite 10). Dadurch ergeben sich zwei Ansätze:

1. Ändern der Definition des Register-Maschinen-Codes

Eine Möglichkeit wäre etwa, die Einführung von Operationen auf lokale Register, Funktionsdefinitionen und Funktionsaufrufen:

- B_i : wie A_i , wirkt aber auf lokales Register i
- T_i : wie S_i , wirkt aber auf lokales Register i
- $[M]_i$: wie $(M)_i$, wirkt aber auf lokales Register i
- $\langle L_j L_j \dots M \rangle_i$: M ist der Funktionsrumpf der Funktion i , Übergabeparameter sind die Register j, j', \dots
- $F_i R_j R_j \dots L_k L_k \dots$: Aufruf der Funktion i , Übergabeparameter sind dabei die globalen Register j, j', \dots und die lokalen Register k, k', \dots . Beim Aufruf der Funktion wird die Referenz auf die Register übergeben, d.h. die Registerinhalte der übergebenen Parameter werden durch die Verwendung der entsprechenden lokalen Register geändert.

2. Statt des Register-Maschinen-Programms wird ein spezieller Interpreter in Register-Maschinen-Code verwendet:

Das erstellte Register-Maschinen-Programm entspricht nicht direkt dem C++-Programm. Es wird ein immer gleiches Programm erstellt und dazu ein Zwischencode der dem eigentlich zu übersetzenden C++-Programm entspricht. Das allgemeine Register-Maschinen-Programm arbeitet dann den Zwischencode ab. Dies entspricht also einer Art universeller Register-Maschine, das das

übersetzte C++-Programm als Daten erhält. Der Zwischencode, der als Registerinhalt an das allgemeine Register-Maschinen-Programm übergeben wird, könnte dann eine Codierung eines erweiterten Register-Maschinen-Codes sein, zum Beispiel der o.g. Vorschlag.

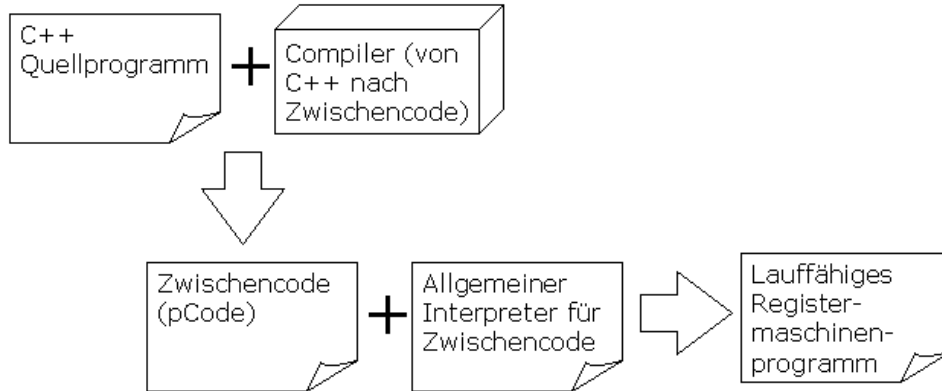


Abbildung III-9: Lösungsvorschlag rekursive Funktionen & lokale Variablen

IV. Das universelle Register-Maschinen-Programm

Das universelle Register-Maschinen-Programm entsteht aus einem Interpreter für Register-Maschinen-Programme in C++. Dieser wird mit dem vorgestellten Compiler übersetzt. Das Ergebnis ist ein Register-Maschinen-Programm, dieses lautet:

(S₅₈)₅₈ A₅₈ A₅₈ A₅₈ A₅₈ A₅₈ A₅₈ A₅₈ A₅₈ A₅₈ (S₄₉)₄₉ (S₂)₂ (S₄₇ A₄₉ A₂)₄₇
(S₂ A₄₇)₂ (S₇)₇ A₇ A₇ A₇ A₇ A₇ A₇ A₇ A₇ A₇ A₇ A₇ (S₂)₂ (S₃)₃ (S₄)₄ (S₆)₆
(S₄₉ A₂ A₃)₄₉ (S₃ A₄₉)₃ (S₇ A₃ A₄)₇ (S₄ A₇)₄ S₃ (S₃ S₂ A₄)₃ (S₄ A₃)₄
A₃ ((S₃ S₂ A₄)₃ (S₄ A₃)₄ A₆)₂ (S₃)₃ (S₆ A₂ A₃)₆ (S₃ A₆)₃ (S₇ A₃
A₄)₇ (S₄ A₇)₄ S₃ (S₃ (S₂ A₆ A₄)₂ (S₄ A₂)₄)₃ (S₂)₂ (S₆ S₄₉)₆ (S₅₈)₅₈
A₅₈ A₅₈ A₅₈ A₅₈ A₅₈ A₅₈ A₅₈ A₅₈ A₅₈ (S₇)₇ A₇ A₇ (S₉)₉ (S₃)₃ (S₄)₄ (S₄₉
A₃ A₄)₄₉ (S₄ A₄₉)₄ (S₇ S₃ A₄)₇ (S₄ A₇)₄ (S₃ A₉)₃ (S₇ A₃ A₄)₇ (S₄ A₇)₄
(S₄₉ S₃ A₄)₄₉ (S₄ A₄₉)₄ (S₃ A₉)₃ A₃ (S₉ S₃)₉ ((S₃)₃ A₉)₃ (S₁)₁
A₁ ((S₇)₇ A₇ A₇ A₇ A₇ A₇ A₇ A₇ A₇ A₇ A₇ (S₂)₂ (S₃)₃ (S₄)₄ (S₄₇ A₂)₄₇
(S₇ A₃ A₄)₇ (S₄ A₇)₄ S₃ (S₃ S₂ A₄)₃ (S₄ A₃)₄ A₃ ((S₃ S₂ A₄)₃ (S₄ A₃)₄
A₄₇)₂ (S₃)₃ (S₁₀)₁₀ (S₁₀)₁₀ (S₄₇ A₁₀)₄₇ (S₁₁)₁₁ (S₁₁)₁₁ (S₄₉ A₁₁)₄₉
(S₁₂)₁₂ (S₁₂)₁₂ (S₅₁ A₁₂)₅₁ (S₁₁)₁₁ (S₂)₂ (S₁₀ A₁₁ A₂)₁₀ (S₂ A₁₀)₂
(S₇)₇ A₇ A₇ A₇ A₇ A₇ A₇ A₇ A₇ A₇ A₇ A₇ (S₂)₂ (S₃)₃ (S₄)₄ (S₆)₆ (S₁₁
A₂ A₃)₁₁ (S₃ A₁₁)₃ (S₇ A₃ A₄)₇ (S₄ A₇)₄ S₃ (S₃ S₂ A₄)₃ (S₄ A₃)₄ A₃ ((S₃
S₂ A₄)₃ (S₄ A₃)₄ A₆)₂ (S₃)₃ (S₆ A₂ A₃)₆ (S₃ A₆)₃ (S₇ A₃ A₄)₇
(S₄ A₇)₄ S₃ (S₃ (S₂ A₆ A₄)₂ (S₄ A₂)₄)₃ (S₂)₂ (S₆ S₁₁)₆ A₅ ((S₅)₅
(S₇)₇ A₇ (S₉)₉ (S₃)₃ (S₄)₄ (S₁₁ A₃ A₄)₁₁ (S₄ A₁₁)₄ (S₇ S₃ A₄)₇ (S₄
A₇)₄ (S₃ A₉)₃ (S₇ A₃ A₄)₇ (S₄ A₇)₄ (S₁₁ S₃ A₄)₁₁ (S₄ A₁₁)₄ (S₃ A₉)₃
A₃ (S₉ S₃)₉ ((S₃)₃ A₉)₃ (A₁₂ (S₇)₇ A₇ A₇ A₇ A₇ A₇ A₇ A₇ A₇ A₇ (S₂)₂
(S₃)₃ (S₄)₄ (S₁₀ A₂)₁₀ (S₇ A₃ A₄)₇ (S₄ A₇)₄ S₃ (S₃ S₂ A₄)₃ (S₄ A₃)₄
A₃ ((S₃ S₂ A₄)₃ (S₄ A₃)₄ A₁₀)₂ (S₃)₃ (S₁₁)₁₁ (S₂)₂ (S₁₀ A₁₁ A₂)₁₀
(S₂ A₁₀)₂ (S₇)₇ A₇ A₇ A₇ A₇ A₇ A₇ A₇ A₇ A₇ A₇ A₇ (S₂)₂ (S₃)₃ (S₄)₄
(S₆)₆ (S₁₁ A₂ A₃)₁₁ (S₃ A₁₁)₃ (S₇ A₃ A₄)₇ (S₄ A₇)₄ S₃ (S₃ S₂ A₄)₃
(S₄ A₃)₄ A₃ ((S₃ S₂ A₄)₃ (S₄ A₃)₄ A₆)₂ (S₃)₃ (S₆ A₂ A₃)₆ (S₃ A₆)₃
(S₇ A₃ A₄)₇ (S₄ A₇)₄ S₃ (S₃ (S₂ A₆ A₄)₂ (S₄ A₂)₄)₃ (S₂)₂ (S₆ S₁₁)₆
A₅ (S₉)₉)₉)₅ (S₁₀ A₄₇)₁₀ (S₁₁ A₄₉)₁₁ (S₁₂ A₅₁)₁₂ (S₃₇)₃₇ (S₃₇)₃₇
(S₅₁ A₃₇)₅₁ (S₃₈)₃₈ (S₃₈)₃₈ (S₄₈ A₃₈)₄₈ (S₃₉)₃₉ (S₃₉)₃₉ (S₅₂ A₃₉)₅₂
(S₄₀)₄₀ (S₄₀)₄₀ (S₅₇ A₄₀)₅₇ (S₄₁)₄₁ (S₄₁)₄₁ (S₅₃ A₄₁)₅₃ (S₄₀)₄₀ (S₂)₂
(S₃₈ A₄₀ A₂)₃₈ (S₂ A₃₈)₂ (S₁₃)₁₃ (S₁₃)₁₃ (S₄₀ A₁₃)₄₀ (S₁₄)₁₄ (S₁₄)₁₄
(S₃₉ A₁₄)₃₉ (S₁₅)₁₅ (S₁₅)₁₅ (S₃₇ A₁₅)₃₇ (S₁₆)₁₆ (S₁₆)₁₆ (S₄₁ A₁₆)₄₁
(S₁₆)₁₆ A₁₆ A₅ ((S₅)₅ (S₂)₂ (S₉)₉ (S₁₅ A₂ A₉)₁₅ (S₂ A₁₅)₂ ((S₁₄)₁₄
(S₂)₂ (S₁₃ A₁₄ A₂)₁₃ (S₂ A₁₃)₂ (S₇)₇ A₇ A₇ A₇ A₇ A₇ A₇ A₇ A₇ A₇
A₇ A₇ (S₂)₂ (S₃)₃ (S₄)₄ (S₆)₆ (S₁₄ A₂ A₃)₁₄ (S₃ A₁₄)₃ (S₇ A₃ A₄)₇
(S₄ A₇)₄ S₃ (S₃ S₂ A₄)₃ (S₄ A₃)₄ A₃ ((S₃ S₂ A₄)₃ (S₄ A₃)₄ A₆)₂ (S₃)₃
(S₆ A₂ A₃)₆ (S₃ A₆)₃ (S₇ A₃ A₄)₇ (S₄ A₇)₄ S₃ (S₃ (S₂ A₆ A₄)₂ (S₄
A₂)₄)₃ (S₂)₂ (S₆ S₁₄)₆ (S₇)₇ (S₉)₉ (S₃)₃ (S₄)₄ (S₁₄ A₃ A₄)₁₄ (S₄
A₁₄)₄ (S₇ S₃ A₄)₇ (S₄ A₇)₄ (S₃ A₉)₃ (S₇ A₃ A₄)₇ (S₄ A₇)₄ (S₁₄ S₃ A₄)₁₄
(S₄ A₁₄)₄ (S₃ A₉)₃ A₃ (S₉ S₃)₉ ((S₃)₃ A₉)₃ (S₁)₁ A₁ (S₁₅ (S₁)₁
(S₉)₉)₉ (S₇)₇ A₇ A₇ A₇ A₇ A₇ A₇ A₇ A₇ A₇ A₇ A₇ (S₂)₂ (S₃)₃ (S₄)₄ (S₁₃
A₂)₁₃ (S₇ A₃ A₄)₇ (S₄ A₇)₄ S₃ (S₃ S₂ A₄)₃ (S₄ A₃)₄ A₃ ((S₃ S₂ A₄)₃
(S₄ A₃)₄ A₁₃)₂ (S₃)₃ (S₇)₇ A₇ A₇ A₇ A₇ A₇ A₇ A₇ A₇ A₇ A₇ A₇ (S₂)₂ (S₃)₃
(S₄)₄ (S₁₆ A₂)₁₆ (S₇ A₃ A₄)₇ (S₄ A₇)₄ (S₃ (S₂ A₁₆ A₄)₂ (S₄ A₂)₄)₃
(S₂)₂ A₅ (S₉)₉)₉)₅ (S₇)₇ A₇ A₇ A₇ A₇ A₇ A₇ A₇ A₇ A₇ A₇ A₇ (S₂)₂ (S₃)₃
(S₄)₄ (S₁₆ A₂)₁₆ (S₇ A₃ A₄)₇ (S₄ A₇)₄ S₃ (S₃ S₂ A₄)₃ (S₄ A₃)₄ A₃ (

$(S_3 S_2 A_4)_3 (S_4 A_3)_4 A_{16})_2 (S_3)_3 (S_{13} A_{40})_{13} (S_{14} A_{39})_{14} (S_{15} A_{37})_{15}$
 $(S_{16} A_{41})_{16} (S_{40})_{40} (S_2)_2 (S_{38} A_{40} A_2)_{38} (S_2 A_{38})_2 (S_2)_2 (S_3)_3 (S_4)$
 $)_4 (S_6)_6 (S_{40} A_2 A_3)_{40} (S_3 A_{40})_3 (S_{41} A_3 A_4)_{41} (S_4 A_{41})_4 S_3 (S_3 S_2 A_4)$
 $)_3 (S_4 A_3)_4 A_3 ((S_3 S_2 A_4)_3 (S_4 A_3)_4 A_6)_2 (S_3)_3 (S_6 A_2 A_3)_6 (S_3 A_6)$
 $)_3 (S_{41} A_3 A_4)_{41} (S_4 A_{41})_4 S_3 (S_3 (S_2 A_6 A_4)_2 (S_4 A_2)_4)_3 (S_2)_2 (S_6$
 $S_{40})_6 (S_2)_2 (S_{40} S_{38} A_2)_{40} (S_2 A_{40})_2 (S_7)_7 A_7 A_7 A_7 A_7 A_7 A_7 A_7 A_7$
 $A_7 (S_2)_2 (S_3)_3 (S_4)_4 (S_{38} A_2)_{38} (S_7 A_3 A_4)_7 (S_4 A_7)_4 (S_3 (S_2 A_{38} A_4$
 $)_2 (S_4 A_2)_4)_3 (S_2)_2 (S_2)_2 (S_{40} A_{38} A_2)_{40} (S_2 A_{40})_2 (S_2)_2 (S_{41} A_{38}$
 $A_2)_{41} (S_2 A_{41})_2 (S_{37} A_{51})_{37} (S_{38} A_{48})_{38} (S_{39} A_{52})_{39} (S_{40} A_{57})_{40} (S_{41}$
 $A_{53})_{41} (S_{49})_{49} (S_1)_1 (S_9)_9)_9 (S_7)_7 A_7 A_7 A_7 (S_9)_9 (S_3)_3 (S_4)_4$
 $(S_{49} A_3 A_4)_{49} (S_4 A_{49})_4 (S_7 S_3 A_4)_7 (S_4 A_7)_4 (S_3 A_9)_3 (S_7 A_3 A_4)_7$
 $(S_4 A_7)_4 (S_{49} S_3 A_4)_{49} (S_4 A_{49})_4 (S_3 A_9)_3 A_3 (S_9 S_3)_9 ((S_3)_3 A_9)_3$
 $(S_1)_1 A_1 ((S_7)_7 A_7 A_7 A_7 A_7 A_7 A_7 A_7 A_7 A_7 (S_2)_2 (S_3)_3 (S_4)_4 (S_{47}$
 $A_2)_{47} (S_7 A_3 A_4)_7 (S_4 A_7)_4 S_3 (S_3 S_2 A_4)_3 (S_4 A_3)_4 A_3 ((S_3 S_2 A_4)_3$
 $(S_4 A_3)_4 A_{47})_2 (S_3)_3 (S_{10})_{10} (S_{10})_{10} (S_{47} A_{10})_{47} (S_{11})_{11} (S_{11})_{11}$
 $(S_{49} A_{11})_{49} (S_{12})_{12} (S_{12})_{12} (S_{51} A_{12})_{51} (S_{11})_{11} (S_2)_2 (S_{10} A_{11} A_2)_{10}$
 $(S_2 A_{10})_2 (S_7)_7 A_7 A_7 A_7 A_7 A_7 A_7 A_7 A_7 A_7 A_7 (S_2)_2 (S_3)_3 (S_4)_4 (S_6$
 $)_6 (S_{11} A_2 A_3)_{11} (S_3 A_{11})_3 (S_7 A_3 A_4)_7 (S_4 A_7)_4 S_3 (S_3 S_2 A_4)_3 (S_4 A_3$
 $)_4 A_3 ((S_3 S_2 A_4)_3 (S_4 A_3)_4 A_6)_2 (S_3)_3 (S_6 A_2 A_3)_6 (S_3 A_6)_3 (S_7 A_3$
 $A_4)_7 (S_4 A_7)_4 S_3 (S_3 (S_2 A_6 A_4)_2 (S_4 A_2)_4)_3 (S_2)_2 (S_6 S_{11})_6 A_5 ($
 $(S_5)_5 (S_7)_7 A_7 (S_9)_9 (S_3)_3 (S_4)_4 (S_{11} A_3 A_4)_{11} (S_4 A_{11})_4 (S_7 S_3 A_4$
 $)_7 (S_4 A_7)_4 (S_3 A_9)_3 (S_7 A_3 A_4)_7 (S_4 A_7)_4 (S_{11} S_3 A_4)_{11} (S_4 A_{11})_4$
 $(S_3 A_9)_3 A_3 (S_9 S_3)_9 ((S_3)_3 A_9)_3 (A_{12} (S_7)_7 A_7 A_7 A_7 A_7 A_7 A_7 A_7 A_7$
 $A_7 A_7 (S_2)_2 (S_3)_3 (S_4)_4 (S_{10} A_2)_{10} (S_7 A_3 A_4)_7 (S_4 A_7)_4 S_3 (S_3 S_2 A_4$
 $)_3 (S_4 A_3)_4 A_3 ((S_3 S_2 A_4)_3 (S_4 A_3)_4 A_{10})_2 (S_3)_3 (S_{11})_{11} (S_2)_2$
 $(S_{10} A_{11} A_2)_{10} (S_2 A_{10})_2 (S_7)_7 A_7 A_7 A_7 A_7 A_7 A_7 A_7 A_7 A_7 A_7 (S_2)_2 (S_3$
 $)_3 (S_4)_4 (S_6)_6 (S_{11} A_2 A_3)_{11} (S_3 A_{11})_3 (S_7 A_3 A_4)_7 (S_4 A_7)_4 S_3 (S_3$
 $S_2 A_4)_3 (S_4 A_3)_4 A_3 ((S_3 S_2 A_4)_3 (S_4 A_3)_4 A_6)_2 (S_3)_3 (S_6 A_2 A_3)_6$
 $(S_3 A_6)_3 (S_7 A_3 A_4)_7 (S_4 A_7)_4 S_3 (S_3 (S_2 A_6 A_4)_2 (S_4 A_2)_4)_3 (S_2)_2$
 $(S_6 S_{11})_6 A_5 (S_9)_9)_9)_5 (S_{10} A_{47})_{10} (S_{11} A_{49})_{11} (S_{12} A_{51})_{12} (S_{42})_{42}$
 $(S_{42})_{42} (S_{51} A_{42})_{51} (S_{43})_{43} (S_{43})_{43} (S_{48} A_{43})_{48} (S_{44})_{44} (S_{44})_{44} (S_{52}$
 $A_{44})_{52} (S_{45})_{45} (S_{45})_{45} (S_{57} A_{45})_{57} (S_{46})_{46} (S_{46})_{46} (S_{53} A_{46})_{53} (S_{45}$
 $)_{45} (S_2)_2 (S_{43} A_{45} A_2)_{43} (S_2 A_{43})_2 (S_{13})_{13} (S_{13})_{13} (S_{45} A_{13})_{45} (S_{14}$
 $)_{14} (S_{14})_{14} (S_{44} A_{14})_{44} (S_{15})_{15} (S_{15})_{15} (S_{42} A_{15})_{42} (S_{16})_{16} (S_{16})_{16}$
 $(S_{46} A_{16})_{46} (S_{16})_{16} A_{16} A_5 ((S_5)_5 (S_2)_2 (S_9)_9 (S_{15} A_2 A_9)_{15} (S_2 A_{15}$
 $)_2 ((S_{14})_{14} (S_2)_2 (S_{13} A_{14} A_2)_{13} (S_2 A_{13})_2 (S_7)_7 A_7 A_7 A_7 A_7 A_7 A_7$
 $A_7 A_7 A_7 A_7 (S_2)_2 (S_3)_3 (S_4)_4 (S_6)_6 (S_{14} A_2 A_3)_{14} (S_3 A_{14})_3 (S_7 A_3$
 $A_4)_7 (S_4 A_7)_4 S_3 (S_3 S_2 A_4)_3 (S_4 A_3)_4 A_3 ((S_3 S_2 A_4)_3 (S_4 A_3)_4 A_6$
 $)_2 (S_3)_3 (S_6 A_2 A_3)_6 (S_3 A_6)_3 (S_7 A_3 A_4)_7 (S_4 A_7)_4 S_3 (S_3 (S_2 A_6 A_4$
 $)_2 (S_4 A_2)_4)_3 (S_2)_2 (S_6 S_{14})_6 (S_7)_7 (S_9)_9 (S_3)_3 (S_4)_4 (S_{14} A_3 A_4$
 $)_{14} (S_4 A_{14})_4 (S_7 S_3 A_4)_7 (S_4 A_7)_4 (S_3 A_9)_3 (S_7 A_3 A_4)_7 (S_4 A_7)_4$
 $(S_{14} S_3 A_4)_{14} (S_4 A_{14})_4 (S_3 A_9)_3 A_3 (S_9 S_3)_9 ((S_3)_3 A_9)_3 (S_1)_1 A_1 ($
 $S_{15} (S_1)_1 (S_9)_9)_9 (S_7)_7 A_7 A_7 A_7 A_7 A_7 A_7 A_7 A_7 A_7 (S_2)_2 (S_3)_3$
 $(S_4)_4 (S_{13} A_2)_{13} (S_7 A_3 A_4)_7 (S_4 A_7)_4 S_3 (S_3 S_2 A_4)_3 (S_4 A_3)_4 A_3 ($
 $(S_3 S_2 A_4)_3 (S_4 A_3)_4 A_{13})_2 (S_3)_3 (S_7)_7 A_7 A_7 A_7 A_7 A_7 A_7 A_7 A_7 A_7$
 $(S_2)_2 (S_3)_3 (S_4)_4 (S_{16} A_2)_{16} (S_7 A_3 A_4)_7 (S_4 A_7)_4 (S_3 (S_2 A_{16} A_4)_2$
 $(S_4 A_2)_4)_3 (S_2)_2 A_5 (S_9)_9)_9)_5 (S_7)_7 A_7 A_7 A_7 A_7 A_7 A_7 A_7 A_7 A_7$
 $(S_2)_2 (S_3)_3 (S_4)_4 (S_{16} A_2)_{16} (S_7 A_3 A_4)_7 (S_4 A_7)_4 S_3 (S_3 S_2 A_4)_3$
 $(S_4 A_3)_4 A_3 ((S_3 S_2 A_4)_3 (S_4 A_3)_4 A_{16})_2 (S_3)_3 (S_{13} A_{45})_{13} (S_{14} A_{44}$
 $)_{14} (S_{15} A_{42})_{15} (S_{16} A_{46})_{16} (S_{45})_{45} (S_2)_2 (S_{46} A_{45} A_2)_{46} (S_2 A_{46})_2$
 $(S_7)_7 A_7 A_7 A_7 A_7 A_7 A_7 A_7 A_7 A_7 A_7 (S_2)_2 (S_3)_3 (S_4)_4 (S_{45} A_2)_{45} (S_7$
 $A_3 A_4)_7 (S_4 A_7)_4 S_3 (S_3 S_2 A_4)_3 (S_4 A_3)_4 A_3 ((S_3 S_2 A_4)_3 (S_4 A_3)_4$

$(S_9)_5 (S_{25})_{25} A_{25} A_{25} A_{25} A_{25} A_{25} A_{25} A_{25} A_{25} A_{25} A_{25} A_{25} A_{25} A_{25} A_{25} A_{25} A_{25} A_{25} A_{25} A_{25}$
 $A_{25} A_{25}$
 $A_{25} A_{25}$
 $A_{25} A_{25}$
 $A_{25} A_{25} A_{25} A_{25} (S_{22})_{22} (S_2)_2 (S_{17} A_{22} A_2)_{17} (S_2 A_{17})_2 (S_2)_2 (S_3)_3$
 $(S_4)_4 (S_6)_6 (S_{22} A_2 A_3)_{22} (S_3 A_{22})_3 (S_{23} A_3 A_4)_{23} (S_4 A_{23})_4 S_3 (S_3$
 $S_2 A_4)_3 (S_4 A_3)_4 A_3 ((S_3 S_2 A_4)_3 (S_4 A_3)_4 A_6)_2 (S_3)_3 (S_6 A_2 A_3)_6$
 $(S_3 A_6)_3 (S_{23} A_3 A_4)_{23} (S_4 A_{23})_4 S_3 (S_3 (S_2 A_6 A_4)_2 (S_4 A_2)_4)_3 (S_2$
 $)_2 (S_6 S_{22})_6 (S_{24})_{24} (S_2)_2 (S_{23} A_{24} A_2)_{23} (S_2 A_{23})_2 (S_7)_7 A_7 A_7 A_7 A_7$
 $A_7 A_7 A_7 A_7 A_7 A_7 (S_2)_2 (S_3)_3 (S_4)_4 (S_{23} A_2)_{23} (S_7 A_3 A_4)_7 (S_4 A_7)_4$
 $S_3 (S_3 S_2 A_4)_3 (S_4 A_3)_4 A_3 ((S_3 S_2 A_4)_3 (S_4 A_3)_4 A_{23})_2 (S_3)_3 (S_7$
 $)_7 A_7 A_7 A_7 A_7 A_7 A_7 A_7 A_7 A_7 (S_2)_2 (S_3)_3 (S_4)_4 (S_{22} A_2)_{22} (S_7 A_3 A_4$
 $)_7 (S_4 A_7)_4 S_3 (S_3 S_2 A_4)_3 (S_4 A_3)_4 A_3 ((S_3 S_2 A_4)_3 (S_4 A_3)_4 A_{22})_2$
 $(S_3)_3 (S_{18})_{18} (S_2)_2 (S_{19} A_{18} A_2)_{19} (S_2 A_{19})_2 (S_7)_7 A_7 A_7 A_7 A_7 A_7 A_7$
 $A_7 A_7 A_7 A_7 (S_2)_2 (S_3)_3 (S_4)_4 (S_6)_6 (S_{18} A_2 A_3)_{18} (S_3 A_{18})_3 (S_7 A_3$
 $A_4)_7 (S_4 A_7)_4 S_3 (S_3 S_2 A_4)_3 (S_4 A_3)_4 A_3 ((S_3 S_2 A_4)_3 (S_4 A_3)_4 A_6$
 $)_2 (S_3)_3 (S_6 A_2 A_3)_6 (S_3 A_6)_3 (S_7 A_3 A_4)_7 (S_4 A_7)_4 S_3 (S_3 (S_2 A_6 A_4$
 $)_2 (S_4 A_2)_4)_3 (S_2)_2 (S_6 S_{18})_6 (S_7)_7 A_7 A_7 A_7 A_7 A_7 A_7 A_7 A_7 A_7 (S_2$
 $)_2 (S_3)_3 (S_4)_4 (S_{19} A_2)_{19} (S_7 A_3 A_4)_7 (S_4 A_7)_4 S_3 (S_3 S_2 A_4)_3 (S_4 A_3$
 $)_4 A_3 ((S_3 S_2 A_4)_3 (S_4 A_3)_4 A_{19})_2 (S_3)_3 (S_{21})_{21} A_5 ((S_5)_5 (S_7)_7$
 $A_7 (S_9)_9 (S_3)_3 (S_4)_4 (S_{18} A_3 A_4)_{18} (S_4 A_{18})_4 (S_7 S_3 A_4)_7 (S_4 A_7)_4$
 $(S_3 A_9)_3 (S_7 A_3 A_4)_7 (S_4 A_7)_4 (S_{18} S_3 A_4)_{18} (S_4 A_{18})_4 (S_3 A_9)_3 A_3$
 $(S_9 S_3)_9 ((S_3)_3 A_9)_3 ((S_{18})_{18} (S_2)_2 (S_{19} A_{18} A_2)_{19} (S_2 A_{19})_2 (S_7$
 $)_7 A_7 A_7 A_7 A_7 A_7 A_7 A_7 A_7 A_7 A_7 (S_2)_2 (S_3)_3 (S_4)_4 (S_6)_6 (S_{18} A_2 A_3$
 $)_{18} (S_3 A_{18})_3 (S_7 A_3 A_4)_7 (S_4 A_7)_4 S_3 (S_3 S_2 A_4)_3 (S_4 A_3)_4 A_3 ((S_3$
 $S_2 A_4)_3 (S_4 A_3)_4 A_6)_2 (S_3)_3 (S_6 A_2 A_3)_6 (S_3 A_6)_3 (S_7 A_3 A_4)_7 (S_4 A_7$
 $)_4 S_3 (S_3 (S_2 A_6 A_4)_2 (S_4 A_2)_4)_3 (S_2)_2 (S_6 S_{18})_6 (S_7)_7 A_7 A_7 A_7 A_7$
 $A_7 A_7 A_7 A_7 A_7 A_7 (S_2)_2 (S_3)_3 (S_4)_4 (S_{19} A_2)_{19} (S_7 A_3 A_4)_7 (S_4 A_7)_4$
 $S_3 (S_3 S_2 A_4)_3 (S_4 A_3)_4 A_3 ((S_3 S_2 A_4)_3 (S_4 A_3)_4 A_{19})_2 (S_3)_3 A_{21} A_5$
 $(S_9)_9 (S_9)_9 (S_{17} A_{47})_{17} (S_{18} A_{49})_{18} (S_{19} A_{57})_{19} (S_{20} A_{50})_{20} (S_{21} A_{51})$
 $)_{21} (S_{22} A_{54})_{22} (S_{23} A_{55})_{23} (S_{24} A_{56})_{24} (S_{32})_{32} (S_{32})_{32} (S_{51} A_{32})_{51}$
 $(S_{33})_{33} (S_{33})_{33} (S_{48} A_{33})_{48} (S_{34})_{34} (S_{34})_{34} (S_{52} A_{34})_{52} (S_{35})_{35} (S_{35}$
 $)_{35} (S_{57} A_{35})_{57} (S_{36})_{36} (S_{36})_{36} (S_{53} A_{36})_{53} (S_{35})_{35} (S_2)_2 (S_{33} A_{35} A_2$
 $)_{33} (S_2 A_{33})_2 (S_{36})_{36} (S_2)_2 (S_{33} A_{36} A_2)_{33} (S_2 A_{33})_2 (S_7)_7 A_7 A_7 A_7$
 $A_7 A_7 A_7 A_7 A_7 A_7 A_7 (S_2)_2 (S_3)_3 (S_4)_4 (S_{36} A_2)_{36} (S_7 A_3 A_4)_7 (S_4 A_7$
 $)_4 (S_3 (S_2 A_{36} A_4)_2 (S_4 A_2)_4)_3 (S_2)_2 (S_7)_7 A_7 A_7 A_7 A_7 A_7 A_7 A_7 A_7$
 $A_7 (S_2)_2 (S_3)_3 (S_4)_4 (S_{36} A_2)_{36} (S_7 A_3 A_4)_7 (S_4 A_7)_4 (S_3 (S_2 A_{36} A_4$
 $)_2 (S_4 A_2)_4)_3 (S_2)_2 A_5 ((S_5)_5 (S_2)_2 (S_9)_9 (S_{32} A_2 A_9)_{32} (S_2 A_{32}$
 $)_2 ((S_{34})_{34} (S_2)_2 (S_{35} A_{34} A_2)_{35} (S_2 A_{35})_2 (S_7)_7 A_7 A_7 A_7 A_7 A_7 A_7$
 $A_7 A_7 A_7 A_7 (S_2)_2 (S_3)_3 (S_4)_4 (S_6)_6 (S_{34} A_2 A_3)_{34} (S_3 A_{34})_3 (S_7 A_3$
 $A_4)_7 (S_4 A_7)_4 S_3 (S_3 S_2 A_4)_3 (S_4 A_3)_4 A_3 ((S_3 S_2 A_4)_3 (S_4 A_3)_4 A_6$
 $)_2 (S_3)_3 (S_6 A_2 A_3)_6 (S_3 A_6)_3 (S_7 A_3 A_4)_7 (S_4 A_7)_4 S_3 (S_3 (S_2 A_6 A_4$
 $)_2 (S_4 A_2)_4)_3 (S_2)_2 (S_6 S_{34})_6 (S_2)_2 (S_9)_9 (S_{34} A_2 A_9)_{34} (S_2 A_{34})_2$
 $(S_1)_1 A_1 (S_{32} (S_1)_1 (S_9)_9)_9 (S_7)_7 A_7 A_7 A_7 A_7 A_7 A_7 A_7 A_7 A_7 (S_2$
 $)_2 (S_3)_3 (S_4)_4 (S_{35} A_2)_{35} (S_7 A_3 A_4)_7 (S_4 A_7)_4 S_3 (S_3 S_2 A_4)_3 (S_4 A_3$
 $)_4 A_3 ((S_3 S_2 A_4)_3 (S_4 A_3)_4 A_{35})_2 (S_3)_3 (S_7)_7 A_7 A_7 A_7 A_7 A_7 A_7 A_7$
 $A_7 A_7 A_7 (S_2)_2 (S_3)_3 (S_4)_4 (S_{36} A_2)_{36} (S_7 A_3 A_4)_7 (S_4 A_7)_4 S_3 (S_3 S_2$
 $A_4)_3 (S_4 A_3)_4 A_3 ((S_3 S_2 A_4)_3 (S_4 A_3)_4 A_{36})_2 (S_3)_3 A_5 (S_9)_9 (S_9)_9)_9)_5$
 $(S_{34})_{34} (S_2)_2 (S_{36} A_{34} A_2)_{36} (S_2 A_{36})_2 (S_7)_7 A_7 A_7 A_7 A_7 A_7 A_7 A_7 A_7$
 $A_7 A_7 (S_2)_2 (S_3)_3 (S_4)_4 (S_6)_6 (S_{34} A_2 A_3)_{34} (S_3 A_{34})_3 (S_7 A_3 A_4)_7$
 $(S_4 A_7)_4 S_3 (S_3 S_2 A_4)_3 (S_4 A_3)_4 A_3 ((S_3 S_2 A_4)_3 (S_4 A_3)_4 A_6)_2 (S_3$
 $)_3 (S_6 A_2 A_3)_6 (S_3 A_6)_3 (S_7 A_3 A_4)_7 (S_4 A_7)_4 S_3 (S_3 (S_2 A_6 A_4)_2 (S_4$

Auffällig sind zum einen die auftretenden Wiederholungen von Additionen „ $A_{31} A_{31} A_{31} A_{31} A_{31} A_{31} A_{31} A_{31} A_{31}$ “, diese werden durch Zuweisung oder Addition einer Konstante zu einem Register erzeugt. Zum anderen fallen aber auch scheinbar sinnlose Initialisierungen von Registern auf: „ $(S_{38})_{38} (S_{38})_{38}$ “. Sie entstehen dadurch, dass bei den meisten Übersetzungen von C++-Codestücken in Registermaschinencode die verwendeten Register zum Teil vorher, aber auch zum Teil hinterher initialisiert werden müssen werden. Durch das Fehlen eines Optimierungslaufs des Compilers werden sie nicht entfernt.

Dieses Register-Maschinen-Programm benötigt für das Programm $(S_1A_2)_{1.1,1,0}$ (Codiert: 14112135,101) 15.681.406.749.438 Schritte für den ersten Interpretationsschritt (um also auf $S_1A_2(S_1A_2)_{1.1,1,0}$ bzw. 1121314112135,101 zu kommen). Dieser „einzelne“ Schritt benötigt, ausgeführt durch den optimierten Interpreter auf einem Intel Celeron 466 Mhz etwa 11 Stunden. (Für das Programm $A_2.1,1,0$ sind 240.593 Schritte notwendig.)

(Das auszuführende Register-Maschinen-Programm wird nach der Codierung von Seite 23 als Startwert zweier Register übergeben. Die Indizes des Programmregisters und des Registerinhaltsregisters sind nach der Übersetzung nicht Eins und Zwei. Dies liegt daran, dass der Compiler die Indizes von 1-9 vorbelegt und erst ab Index 10 die Variablen legt. Mit einem gewöhnlichen Texteditor kann dies aber leicht durch „Suchen & Ersetzen“ bereinigt werden.)

V. Zusammenfassung und Ausblick

Ziel dieser Arbeit war es einen Weg zu finden, ein universelles Register-Maschinen-Programm herzustellen. Die Vermutung, dass dies über eine Kombination von Interpreter und Compiler möglich ist, hat sich als richtig erwiesen. Der in C++ implementierte Interpreter war einfach genug gebaut, um durch den hier vorgestellten Compiler von einer reduzierten C++-Syntax in Register-Maschinen-Sprache übersetzt zu werden.

Der nächste Schritt wäre es eine bessere Codierung der Register-Maschinen-Programme zu finden, um so das universelle Register-Maschinen-Programm zu vereinfachen – also zu verkürzen. Die vorgestellte Codierung enthält im Befehlscode nur vier von zehn möglichen Codes, im Register-Inhalt sogar nur zwei von zehn möglichen. Denkbar wäre etwa für den Register-Inhalt die auch von Penrose benutzte expandierte Binärdarstellung.

Um die Laufzeit für das universelle Register-Maschinen-Programm zu verringern, wäre auch eine Verbesserung des optimierten Interpreters möglich. Der bereits beschrittene Weg des „Pattern-Matching“, der Ausführung von ganzen Befehlsfolgen in einem einzigen Schritt, kann sicher noch weiter ausgebaut werden.

Der Ausbau des Compilers wurde schon angesprochen: rekursive Funktionen und lokale Variablen sollten hier das Hauptziel sein. Ein Ausbau in Richtung ANSI C++ scheint ansonsten nicht schwierig, da der Verwendung von LEX und YACC nichts im Wege steht und alle weiteren Konstrukte (wie etwa Zählschleifen) sich zu vorhandenen Konstrukten umformulieren lassen.

VI. Literatur- & Quellenverzeichnis

- Albert, J. und Ottmann, Th.: Automaten, Sprachen und Maschinen für Anwender. Reihe Informatik / 38. BI Wissenschaftsverlag 1990.
- Church, Alonzo: An unsolvable problem in elementary number theory. American Journal of Mathematics 1936. Vol. 58. Seite 345-363.
- Cutland, N.: Computability: an introduction to recursive function theory. Cambridge University Press 1980.
- Penrose, Roger: Computerdenken - Des Kaisers neue Kleider oder die Debatte um die Künstliche Intelligenz, Bewußtsein und die Gesetze der Physik. Spektrum der Wissenschaft 1991.
- Porr, Bernd: Von der Differenzmaschine zum PC. Webseite und Hyperlinks.
<http://www.neurop.ruhr-uni-bochum.de/~porr/compgesch/compgesch.html>
- Reischuk, K. Rüdiger: Komplexitätstheorie. B. G. Teubner 1999.
- Schnitzspan, H.: Vorlesung AAT. Für Studenten der Informatik, Fachsemester 5. Wintersemester 1998/99.
- Shepherdson, J.C. & Sturgis, H.E.: Computability of Recursive Functions. Journal of the ACM 1963. Vol 10. Seite 217-255. (*Achtung: z.T. in Quellenangaben auch „Sheperdson“ statt „Shepherdson“*)
- Stroustrup, B.: The C++ Programming Language. Addison Wesley Publishing Company 1997.
- Turing, Alan M.: On Computable Numbers with an Application to the Entscheidungsproblem. Proc. London Math. Soc. 1935. Vol. (2) 42. Seite 230-265.
- Wirth, N.: Compilerbau: eine Einführung. B. G. Teubner 1986.
- Zuse, Horst: History of Computing. Webseite und Hyperlinks.
<http://irb.cs.tu-berlin.de/~zuse/>

VII. Verzeichnis der Abbildungen und Tabellen

Abbildung II-1 Schema der Register-Maschine	11
Abbildung II-2: Syntax des Register-Maschinen-Codes	12
Abbildung II-3: Schema des Interpreters.....	21
Abbildung II-4: Schema des optimierten Interpreters	22
Abbildung II-5: Bildschirmausdruck des Laufzeitoptimierten Interpreters	23
Abbildung II-6: Struktogramm des Additions-Teils des Interpreters.....	25
Abbildung III-1: Struktur SObject.....	34
Abbildung III-2: Beispiel für die Verwendung der Struktur SObject	34
Abbildung III-3: Struktogramm der Multiplikation „*=“	36
Abbildung III-4: Struktogramm der Division „/=“	37
Abbildung III-5: Struktogramm der Modulo Operation „%=“	37
Abbildung III-6: Struktogramm des Tests auf Gleichheit	39
Abbildung III-7: Übersetzung der if- Anweisung	40
Abbildung III-8: Übersetzung der while-Anweisung	41
Abbildung III-9: Lösungsvorschlag rekursive Funktionen & lokale Variablen.....	43
Tabelle II-1: Die Funktion g, die aus altem Stapel und altem Tupel den neuen Stapel und das neue Tupel berechnet.....	14
Tabelle II-2: Klasse CRegister.....	20
Tabelle II-3: Umwandlungstabelle des Befehlscodes für den Interpreter	24
Tabelle III-1: Klasse CErrorCompiler	26
Tabelle III-2: Klasse CToken	26
Tabelle III-3: Klasse CParse	28
Tabelle III-4: Struktur SObject.....	29
Tabelle III-5: EBNF-Darstellung der verwendeten C++ Syntax	32
Tabelle III-6: Festlegung der Hilfsregister	33

VIII. Glossar

Operationen

Gemeint sind damit alle vom Compiler implementierten Operationen. Dies sind die unären Operationen "++" und "--", die binären Operationen (Addition) "+=", (Subtraktion) "-=", (Multiplikation) "*=", (Division) "/=", (Modulo) "%=" und die Operationen (Addition) "+", (Subtraktion) "-", (Multiplikation) "*", (Division) "/", (Modulo) "%"

Funktion g

Die Funktion, die aus dem alten Stapel und alten Tupel den neuen Stapel und das neue Tupel berechnet. Sie kann durch eine Tabelle angegeben werden (siehe auch Seite 14).

long

(C++) Zahlen-Typ, der ganze Zahlen von -2147483646 bis +2147483647 aufnehmen kann.

CRegister

(C++) Klasse, um "unendlich" große Zahlen darzustellen (siehe auch Klassenübersicht Seite 20).

CParse

(C++) Klasse, die den Parse & Compiler von C++ nach Register-Maschinen-Code implementiert (siehe auch Klassenübersicht Seite 28).

CToken

(C++) Klasse, um C++ Token aus einer Datei einzulesen (siehe auch Klassenübersicht Seite 26).

Unäre Zahlen

Darstellungsart von Zahlen zur Basis 1. Beispiel: dezimal 6 entspricht unär 111111.

Weitere übliche Darstellungen: binär (Basis 2) (Beispiel: 6 dezimal entspricht 110 binär), hexadezimal (Basis 16) (Beispiel: 26 dezimal entspricht 1A hexadezimal)

Register-Maschinen-Befehl

Einzelner Befehl für eine Register-Maschine. Es gibt 3 Register-Maschinen-Befehle (nach Ottmann und Albert¹⁸):

1. A_i: Addition
2. S_i: Subtraktion

¹⁸ Albert, J. und Ottmann, Th.: Automaten, Sprachen und Maschinen für Anwender.

3. $(M)_i$: Schleife

Zur genauen Definition siehe auch „Definition der Register-Maschine“.

Register-Maschinen-Code

Geordnete Folge von Register-Maschinen-Befehlen

Register-Maschinen-Programm

Register-Maschinen-Code und Registerinhalte zusammen

Beispiel: $A_1 \ S_2 \ . \ 1, 2, 0$

IX. Danksagungen

Vor allen anderen möchte ich mich bei Daniela Groß für die Geduld bedanken, die sie während der ganzen Zeit mit mir hatte. Danke für den Kommentar „Mach’s nochmal“ zur ersten Version der Einleitung.

Die gute Strukturierung der „Arbeitsanweisung“ für diese Diplomarbeit von Herrn Schnitzspan hat mir sehr geholfen mich im Chaos zurechtzufinden. Die Treffen mit ihm waren sehr fruchtbar, alleine auch schon dadurch, dass er mir geholfen hat, meine Gedanken zu ordnen.

Besonders erwähnen möchte ich Marc Pfetsch, der den Inhalt verstehen konnte und mir durch Korrekturlesen einige Missverständnisse erspart hat. Yves Lorat, Tilo Reinhardt, Siegfried Kühn, Klaus Raabe und noch einigen anderen gilt mein Dank für die blöden Kommentare, die mir doch hin und wieder nützlich waren.

Ein Hund der Gras frisst, ist ein Schaf.
T. Reinhardt, August 2000

Nenns doch dynamisches Funktionscodeparameterablagemodell.
S. Kühn, September 2000

Das heißt nicht „Interpreter“, das heißt „Interpret“!
D. Groß, September 2000

Erkenne dich selbst.
(gr. "gnóthi seautón") Inschrift am Apollontempel in Delphoi, die von Cheilon, einem der [sieben Weisen](#), stammen soll

Don't worry about people stealing your ideas. If your ideas are any good, you'll have to ram them down people's throats.
Howard Aiken (1900-1973)

Being a language, mathematics may be used not only to inform but also, among other things, to seduce.
Benoit Mandelbrot (1924-)

In mathematics you don't understand things. You just get used to them.
John von Neumann (1903-1957)

Knowing what is big and what is small is more important than being able to solve partial differential equations.
Stan Ulam (1909-1984)

Any good idea can be stated in fifty words or less.
Stan Ulam (1909-1984)

X. Anhang

1. Klasse CRegister

1.1. Vollständige Übersichtstabelle der Klasse

CRegister	
Klasse für große natürliche Zahlen. Stellt alle gängigen Rechenoperationen zur Verfügung.	
Methode/ Eigenschaft	Beschreibung
CRegister	Konstruktor
~CRegister	Destruktor
operator=	Kopiert das gegebene Register auf das aktuelle, sodass deren Werte gleich sind
operator+	Addition von zwei Registern: $R_{\text{this}} + R_x$, liefert das Ergebnis als neues Register zurück
operator*	Multiplikation von zwei Registern: $R_{\text{this}} * R_x$, liefert das Ergebnis als neues Register zurück
operator-	Subtraktion von zwei Registern: $R_{\text{this}} - R_x$, liefert das Ergebnis als neues Register zurück
operator/	Division von zwei Registern: R_{this} / R_x , liefert das Ergebnis als neues Register zurück
operator%	Modulo von zwei Registern: $R_{\text{this}} \% R_x$, liefert das Ergebnis als neues Register zurück
operator++	Inkrementiert das aktuelle Register (Addition von 1)
operator--	Dekrementiert das aktuelle Register (Subtraktion von 1)
operator+=	Addiert das übergebene Register zum aktuellen hinzu
operator-=	Subtrahiert das übergebene Register vom aktuellen
operator*= operator/=	Multipliziert das übergebene Register mit dem aktuellen, das Ergebnis wird im aktuellen Register abgelegt
operator/=	Dividiert das übergebene Register mit dem aktuellen, das Ergebnis wird im aktuellen Register abgelegt
operator%= operator%=	Führt die Modulo Operation mit dem gegebenen Register und dem aktuellen Register durch, das Ergebnis wird im aktuellen Register abgelegt

CRegister	
Klasse für große natürliche Zahlen. Stellt alle gängigen Rechenoperationen zur Verfügung.	
Methode/ Eigenschaft	Beschreibung
long	Konvertiert das aktuelle Register zum Datentyp long. Ist die Zahl zu groß wird -1 zurückgeliefert.
iszero	Prüfung auf Null
toChar	Konvertiert das aktuelle Register zum Datentyp char *
char * m_pcValue	Zeichenkette, die den Registerinhalt enthält
long m_lCurrentSize	Länge des Speicherbereichs. Der Speicherbereich wird zu Anfang mit 100, dann 1.000, 10.000, etc. Bytes belegt, um zu häufige Speicheranforderungen und -freigaben zu verhindern.
newSpace	Fordert neuen Speicher an. Falls der aktuelle nicht ausreicht wird neuer Speicher belegt, der alte Inhalt kopiert und danach freigegeben. Verwendet m_lCurrentSize um die Größe des neu belegten Speichers zu sichern.
mul10	Multiplikation des aktuellen Registers mit 10. (Verwendet Zeichenkettenoperationen: hängt ein Nullzeichen an)
div10	Division des aktuellen Registers durch 10. (Verwendet Zeichenkettenoperationen: entfernt letzte Ziffer)
mod10	Modulo 10 des aktuellen Registers. (Verwendet Zeichenkettenoperationen: entfernt alle Ziffern bis auf die letzte)
RemoveLeadingZero	Bereinigungsfunktion, die führende Nullen entfernt. Diese Nullen werden durch Subtraktionen oder Divisionen hervorgerufen

1.2. CRegister.cpp

```
#include "register.h"
#include <stdlib.h>
#include <string.h>
#include <math.h>
#include <iostream.h>
```

```
// -----
// Constructor & Destructor, cast-operators
// -----
```

```
CRegister::CRegister() {
    m_lCurrentSize = 0;
    m_pcValue = NULL;
```

```

    newSpace(2);
    memset(m_pcValue,0,2);
    strcpy(m_pcValue, "0");
}

CRegister::CRegister(long l) {
    char buffer[20];
    ltoa(l,buffer,10);
    m_lCurrentSize = 0;
    m_pcValue = NULL;

    newSpace(strlen(buffer)+1);
    memset(m_pcValue,0,strlen(buffer)+1);
    strcpy(m_pcValue,buffer);
}

CRegister::CRegister(const CRegister & rInit) {
    m_lCurrentSize = 0;
    m_pcValue = NULL;

    newSpace(strlen(rInit.m_pcValue)+1);
    memset(m_pcValue, 0, strlen(rInit.m_pcValue));
    strcpy(m_pcValue, rInit.m_pcValue);
}

CRegister::CRegister(char * pcInitialValue) {
    m_lCurrentSize = 0;
    m_pcValue = NULL;

    newSpace(strlen(pcInitialValue)+1);
    memset(m_pcValue,0,strlen(pcInitialValue)+1);
    strcpy(m_pcValue, pcInitialValue);
}

CRegister::~CRegister() {
    if (m_pcValue != NULL) {
        delete [] m_pcValue; }
    if ((m_pcValue == NULL) && (m_lCurrentSize > 0)) {
        throw CErrorMemory();
    }
    m_pcValue = NULL;
}

////////////////////////////////////
// Convert to type long
////////////////////////////////////
CRegister::operator long(){
    if (strlen(m_pcValue) < 10) {
        return atol(m_pcValue);
    } else
        return -1L;
}

////////////////////////////////////
// Convert to type char *
////////////////////////////////////
char * CRegister::toChar() {
    return m_pcValue;
}

// -----
// internal usage functions (memory, etc.)
// -----

////////////////////////////////////
// allocate new char array if needed, when allocate, do it in blocks
////////////////////////////////////
void CRegister::newSpace(long lNewSize) {
    char * pcNew = NULL;
    long lCurrentSize = m_lCurrentSize;

    if (lNewSize >= lCurrentSize) {
        if (lCurrentSize == 0)
            lCurrentSize = MINIMUM_ALLOCATE;
        while (lNewSize >= lCurrentSize) {
            lCurrentSize *= 10;
        }
        pcNew = new char [lCurrentSize];
    }
}

```

```

        if (pcNew == NULL)
            throw CErrorMemory();
        else {
            memset(pcNew, 0, lCurrentSize * sizeof(char));
            if (m_pcValue != NULL) {
                strcpy(pcNew, m_pcValue);
                delete [] m_pcValue;
            }
            m_pcValue = pcNew;
            m_lCurrentSize = lCurrentSize;
        }
    }
}

////////////////////////////////////
// remove leading zeros in given char array at beginning of array
////////////////////////////////////
void CRegister::RemoveLeadingZero(char * & pcValue) {

    if ((pcValue[0] == '0') && (strlen(pcValue) > 1)) {
        unsigned long lRemove = 0;

        while (pcValue[lRemove] == '0') {
            lRemove++;
            if (lRemove >= strlen(pcValue))
                break;
        }

        if (lRemove > 0) {
            if (lRemove >= strlen(pcValue))
                lRemove = strlen(pcValue) - 1;

            memmove(pcValue, pcValue+lRemove, (strlen(pcValue) - lRemove + 1) * sizeof(char));
        }
    }
}

// -----
// check-operator, copy-operator, etc.
// -----

////////////////////////////////////
// check if value is zero
////////////////////////////////////
bool CRegister::iszero() {
    return ((strlen(m_pcValue) == 1) && (m_pcValue[0] == '0'));
}

////////////////////////////////////
// copy operator
////////////////////////////////////
void CRegister::operator=(const CRegister & r) {
    if (r.m_pcValue != m_pcValue) {
        newSpace(strlen(r.m_pcValue));
        strcpy(m_pcValue, r.m_pcValue);
    }
}

////////////////////////////////////
// copy operator (long -> CRegister)
////////////////////////////////////
void CRegister::operator=(const long l) {
    newSpace((long)log(l)+2);
    ltoa(l, m_pcValue, 10);
}

// -----
// ADD
// -----

////////////////////////////////////
// increase value by 1
////////////////////////////////////
CRegister & CRegister::operator++(int) {
    long i = strlen(m_pcValue) - 1;

```

```

if (i < 0) {
    newSpace(2);
    strcpy(m_pcValue, "0");
    i = strlen(m_pcValue);
}

if (i >= 0) {
    do {
        char c = m_pcValue[i];

        if (c != '9') {
            m_pcValue[i]++;
            break;
        }
        else {
            m_pcValue[i] = '0';
            i--;
        }
    } while (i >= 0);
    if (i < 0) {
        newSpace(strlen(m_pcValue) + 1 + 1);
        memmove(m_pcValue+1, m_pcValue, (strlen(m_pcValue)+1)*sizeof(char));
        m_pcValue[0] = '1';
    }
}

return *this;
}

////////////////////////////////////
// add constant to value
////////////////////////////////////
CRegister CRegister::operator+(const CRegister & a2){
    CRegister rR;
    long l1 = strlen(m_pcValue);
    long l2 = strlen(a2.m_pcValue);
    long l = 0;
    long i = 0;
    long v1 = 0;
    long v2 = 0;
    long u = 0;
    char *pcR1 = new char [l1+l2+1];

    memset(pcR1, 0, l1+l2+1);

    if (l1 > l2) l = l1; else l = l2;

    for(i=1;i<=l;i++) {
        if (l1 - i >= 0) v1 = m_pcValue[l1-i] - '0'; else v1 = 0;
        if (l2 - i >= 0) v2 = a2.m_pcValue[l2-i] - '0'; else v2 = 0;

        if (v2 + v1 + u < 10) {
            pcR1[l-i] = (char)('0' + (char)(v2 + v1 + u));
            u = 0;
        } else {
            pcR1[l-i] = (char)('0' + (char)(v2 + v1 + u - 10));
            u = (v2 + v1 + u) / 10;
        }
    }

    if (u > 0) {
        rR.newSpace(strlen(pcR1)+2);
        rR.m_pcValue[0] = (char)('0' + (char)u);
        rR.m_pcValue[1] = 0;
        strcat(rR.m_pcValue, pcR1);
    } else {
        rR.newSpace(strlen(pcR1)+1);
        strcpy(rR.m_pcValue, pcR1);
    }

    delete [] pcR1;

    return rR;
}

////////////////////////////////////
// add "a" to "this"

```

```

////////////////////////////////////
CRegister & CRegister::operator+=(const CRegister & a) {
    long l1 = strlen(m_pcValue);
    long l2 = strlen(a.m_pcValue);
    long l = 0;
    long i = 0;
    long v1 = 0;
    long v2 = 0;
    long u = 0;
    char *pcR1 = new char [l1+l2+1];

    memset(pcR1, 0, l1+l2+1);

    if (l1 > l2) l = l1; else l = l2;

    for(i=1;i<=l;i++) {
        if (l1 - i >= 0) v1 = m_pcValue[l1-i] - '0'; else v1 = 0;
        if (l2 - i >= 0) v2 = a.m_pcValue[l2-i] - '0'; else v2 = 0;

        if (v2 + v1 + u < 10) {
            pcR1[l-i] = (char)('0' + (char)(v2 + v1 + u));
            u = 0;
        } else {
            pcR1[l-i] = (char)('0' + (char)(v2 + v1 + u - 10));
            u = (v2 + v1 + u) / 10;
        }
    }

    if (u > 0) {
        newSpace(strlen(pcR1)+2);
        m_pcValue[0] = (char)('0' + (char)u);
        m_pcValue[1] = 0;
        strcat(m_pcValue, pcR1);
    } else {
        newSpace(strlen(pcR1)+1);
        strcpy(m_pcValue, pcR1);
    }

    delete [] pcR1;

    return *this;
}

////////////////////////////////////
// add "a" (long) to "this"
////////////////////////////////////
CRegister & CRegister::operator+=(const long a) {
    *this += CRegister(a);
    return *this;
}

// -----
// SUBTRACT
// -----

////////////////////////////////////
// decrease value by 1
////////////////////////////////////
CRegister & CRegister::operator--(int) {
    long i = strlen(m_pcValue) - 1;

    if (i < 0) {
        newSpace(2);
        strcpy(m_pcValue, "0");
        i = strlen(m_pcValue);
    }

    if (i >= 0) {
        if (!iszero()) {
            do {
                char c = m_pcValue[i];

                if (c != '0') {
                    m_pcValue[i]--;
                    break;
                }
            }
            else {

```

```

                m_pcValue[i] = '9';
                i--;
            }
        } while (i >= 0);
    }
}

RemoveLeadingZero(m_pcValue);

return *this;
}

////////////////////////////////////
// compute "this" - "a" and return value as new object
////////////////////////////////////
CRegister CRegister::operator-(const CRegister & a){
    CRegister rR(m_pcValue);
    long l1 = strlen(a.m_pcValue);
    long l2 = strlen(m_pcValue);
    long l = 0;
    long i = 0;
    long v1 = 0;
    long v2 = 0;
    long u = 0;
    char *pcR = new char [l1+l2+1];

    memset(pcR, 0, l1+l2+1);

    if (l1 > l2) l = l1; else l = l2;

    for(i=l-1;i>=0;i--) {
        l1--;
        l2--;
        if (l1 >= 0) v1 = a.m_pcValue[l1] - '0'; else v1 = 0;
        if (l2 >= 0) v2 = m_pcValue[l2] - '0'; else v2 = 0;

        if (v2 - v1 - u >= 0) {
            pcR[i] = (char)('0' + (char)(v2 - v1 - u));
            u = 0;
        } else {
            pcR[i] = (char)('0' + (char)(v2 - v1 - u + 10));
            u = 1;
        }
    }

    RemoveLeadingZero(pcR);
    rR.newSpace(strlen(pcR)+1);

    strcpy(rR.m_pcValue, pcR);

    delete [] pcR;
    return rR;
}

////////////////////////////////////
// subtract "a" from "this"
////////////////////////////////////
CRegister & CRegister::operator-=(const CRegister & a) {
    long l1 = strlen(a.m_pcValue);
    long l2 = strlen(m_pcValue);
    long l = 0;
    long i = 0;
    long v1 = 0;
    long v2 = 0;
    long u = 0;
    char *pcR = new char [l1+l2+1];

    memset(pcR, 0, l1+l2+1);

    if (l1 > l2) {
        m_pcValue[0] = '0';
        m_pcValue[1] = 0;
    } else {
        if (l1 > l2) l = l1; else l = l2;

        for(i=l-1;i>=0;i--) {
            l1--;
            l2--;

```

```

        if (l1 >= 0) v1 = a.m_pcValue[l1] - '0'; else v1 = 0;
        if (l2 >= 0) v2 = m_pcValue[l2] - '0'; else v2 = 0;

        if (v2 - v1 - u >= 0) {
            pcR[i] = (char)('0' + (char)(v2 - v1 - u));
            u = 0;
        } else {
            pcR[i] = (char)('0' + (char)(v2 - v1 - u + 10));
            u = 1;
        }
    }

    if (u > 0) {
        m_pcValue[0] = '0';
        m_pcValue[1] = 0;
    } else {

        newSpace(strlen(pcR)+1);

        strcpy(m_pcValue, pcR);
        RemoveLeadingZero(m_pcValue);
    }
}
delete [] pcR;
return *this;
}

////////////////////////////////////
// subtract "a" (long) from "this"
////////////////////////////////////
CRegister & CRegister::operator--(const long a) {
    *this -= CRegister(a);
    return *this;
}

// -----
// MULTIPLY
// -----

////////////////////////////////////
// compute "this" * "m" and return value as new object
////////////////////////////////////
CRegister CRegister::operator*(const CRegister & m){
    CRegister * rReturn = new CRegister;
    CRegister r1,r2,r3;
    long p1,p2,pTemp;
    long i1,i2;
    long l1 = strlen(m_pcValue);
    long l2 = strlen(m.m_pcValue);

    p1 = 0;
    for (i1=l1-1;i1>=0;i1--) {
        p2 = 0;
        for(i2=l2-1;i2>=0;i2--) {
            pTemp = p1 + p2;
            r1 = (long)(m_pcValue[i1] - '0') * (long)(m.m_pcValue[i2] - '0');
            while (pTemp > 0) { r1.mull0(); pTemp--; };
            r2 = r2 + r1;
            p2++;
        }
        p1++;
    }
    *rReturn = r2;

    RemoveLeadingZero(rReturn->m_pcValue);

    return *rReturn;
}

////////////////////////////////////
// multiply "m" with "this"
////////////////////////////////////
CRegister & CRegister::operator*=(const CRegister & m) {
    CRegister r1,r2;
    long p1,p2,pTemp;
    long i1,i2;
    long l1 = strlen(m_pcValue);
    long l2 = strlen(m.m_pcValue);

```

```

char * pM = new char [strlen(m.m_pcValue)+1];

memset(pM, 0, (strlen(m.m_pcValue)+1)*sizeof(char));
strcpy(pM, m.m_pcValue);

while((pM[strlen(pM)-1]) == '0') {
    mull10();
    pM[strlen(pM)-1] = 0;
}

if ((strlen(pM) != 1) && (pM[0] == '1')) {
    p1 = 0;
    for (i1=l1-1;i1>=0;i1--) {
        p2 = 0;
        for(i2=l2-1;i2>=0;i2--) {
            pTemp = p1 + p2;
            r1 = ((long)(m_pcValue[i1] - '0') * (long)(pM[i2] - '0'));
            while (pTemp > 0) { r1.mull10(); pTemp--; };
            r2 = r2 + r1;
            p2++;
        }
        p1++;
    }

    newSpace(strlen(r2.m_pcValue)+1);
    strcpy(m_pcValue, r2.m_pcValue);
}

RemoveLeadingZero(m_pcValue);

return *this;
}

////////////////////////////////////
// compute "this" = "this" * "m"
////////////////////////////////////
CRegister & CRegister::operator*=(const long & m) {
    long m2 = m;

    while ((m2 % 10) == 0) {
        m2 /= 10;
        mull10();
    }

    RemoveLeadingZero(m_pcValue);

    if (m2 != 1)
        *this *= CRegister(m2);

    return *this;
}

////////////////////////////////////
// compute "this" = "this" * 10
////////////////////////////////////
CRegister & CRegister::mull10() {
    if (!iszero()) {
        newSpace(strlen(m_pcValue)+2);
        strcat(m_pcValue, "0");
    }

    return *this;
}

// -----
// DIVIDE
// -----

////////////////////////////////////
// compute "this" / "q" and return value as new object
////////////////////////////////////
CRegister CRegister::operator/(const CRegister & q){
    CRegister * rReturn = new CRegister();
    CRegister r1(m_pcValue);
    CRegister r3;
    CRegister r2;
    CRegister r4;

```

```

long    l1 = strlen(m_pcValue);
long    l2 = strlen(q.m_pcValue);
long    ld = 0;

while (!r1.iszero()) {
    l1 = strlen(r1.m_pcValue);
    ld = l1 - l2;
    if (ld < 0) {
        r1 = 0;
    } else {
        ld--;
        if (ld > 0) {
            r2 = q;
            r4 = 1;
            while (ld > 0) { r2.mull0(); r4.mull0(); ld--; };
            r1 = r1 - r2; r3 = r3 + r4;
        } else {
            r1 = r1 - q; r3++;
        }
    }
}

*rReturn = r3;

RemoveLeadingZero(rReturn->m_pcValue);

return *rReturn;
}

////////////////////////////////////
// divide "this" by "q"
////////////////////////////////////
CRegister & CRegister::operator/=(const CRegister & q) {
    if ((q.m_pcValue[strlen(q.m_pcValue)-1] == '0') && (strlen(q.m_pcValue) > 1)) {
        long l = strlen(q.m_pcValue) - 1;
        while (q.m_pcValue[l] == 0) {
            l--;
            div10();
        }
        if ((q.m_pcValue[l] != '1') || (l != 0)) {
            char * qTemp = new char [l+1];
            strncpy(qTemp, q.m_pcValue, l);
            qTemp[l] = 0;

            *this /= CRegister(qTemp);

            delete [] qTemp;
        }
    } else if ((q.m_pcValue[0] == '1') && (strlen(q.m_pcValue) == 1)) {
    } else {
        CRegister r1;
        CRegister r3;
        CRegister r2;
        CRegister r4;
        long    l1 = strlen(m_pcValue);
        long    l2 = strlen(q.m_pcValue);
        long    ld = 0;

        r1 = *this;

        while (!r1.iszero()) {
            l1 = strlen(r1.m_pcValue);
            ld = l1 - l2;
            if (ld < 0) {
                r1 = 0;
            } else {
                ld--;
                if (ld > 0) {
                    r2 = q;
                    r4 = 1;
                    while (ld > 0) { r2.mull0(); r4.mull0(); ld--; };
                    r1 = r1 - r2; r3 = r3 + r4;
                } else {
                    r1 = r1 - q; r3++;
                }
            }
        }
    }
}

```

```

        newSpace(strlen(r3.m_pcValue)+1);
        strcpy(m_pcValue, r3.m_pcValue);

        RemoveLeadingZero(m_pcValue);
    }
    return *this;
}

////////////////////////////////////
// compute "this" = "this" / "q"
////////////////////////////////////
CRegister & CRegister::operator/=(const long & q) {
    long q2 = q;

    while ((q2 % 10) == 0) {
        q2 /= 10;
        div10();
    }

    RemoveLeadingZero(m_pcValue);

    if (q2 != 1)
        *this /= CRegister(q2);

    return *this;
}

////////////////////////////////////
// compute "this" = "this" / 10
////////////////////////////////////
CRegister & CRegister::div10() {
    if (strlen(m_pcValue) > 1) {
        m_pcValue[strlen(m_pcValue)-1] = 0;
    } else {
        m_pcValue[0] = '0';
    }

    return *this;
}

// -----
// MODULO
// -----

////////////////////////////////////
// compute "this" % "q" and return value as new object
////////////////////////////////////
CRegister CRegister::operator%(const CRegister & q){
    CRegister * prReturn = new CRegister();
    CRegister r1(m_pcValue);
    CRegister r2(m_pcValue);
    r1 /= q;
    r1 *= q;
    *prReturn = r2 - r1;

    RemoveLeadingZero(prReturn->m_pcValue);

    return *prReturn;
}

////////////////////////////////////
// compute "this" = "this" % "q"
////////////////////////////////////
CRegister & CRegister::operator%=(const CRegister & q) {
    if (q.m_pcValue[0] == '1') {
        long l = 1;
        bool bZero = true;

        for(l=1;l<(long)strlen(q.m_pcValue);l++) {
            if (q.m_pcValue[l] != '0') {
                bZero = false;
                break;
            }
        }

        if (bZero && (strlen(m_pcValue) >= strlen(q.m_pcValue)) && (q.m_pcValue[0] == '1')) {
            long lRemove = strlen(m_pcValue) - strlen(q.m_pcValue) + 1;

```

```

        memmove(m_pcValue, m_pcValue + lRemove, strlen(q.m_pcValue));
    } else if (bZero && (strlen(m_pcValue) < strlen(q.m_pcValue)) && (q.m_pcValue[0] == '1'))
    {
        //
    } else {
        CRegister r(m_pcValue);
        r = r % q;
        newSpace(strlen(r.m_pcValue)+1);
        strcpy(m_pcValue,r.m_pcValue);
    }
}

RemoveLeadingZero(m_pcValue);

return *this;
}

/////////////////////////////////////////////////////////////////
// compute "this" = "this" % "q"
/////////////////////////////////////////////////////////////////
CRegister & CRegister::operator%=(const long & q) {
    long q2 = q;
    long l = 0;

    while ((q2 % 10) == 0) {
        q2 /= 10;
        l++;
    }

    if (q2 == 1) {
        if (l < (long)strlen(m_pcValue)) {
            memmove(m_pcValue, m_pcValue+(strlen(m_pcValue)-l)*sizeof(char), l+1);
        }
    } else {
        q2 = q;
    }

    RemoveLeadingZero(m_pcValue);

    return *this;
}

/////////////////////////////////////////////////////////////////
// compute "this" = "this" % 10
/////////////////////////////////////////////////////////////////
CRegister & CRegister::mod10() {
    if (!iszero()) {
        if (strlen(m_pcValue) > 1) {
            m_pcValue[0] = m_pcValue[strlen(m_pcValue)-1];
            m_pcValue[1] = 0;
        }
    }
    return *this;
}

```

1.3. CRegister.h

```

#ifndef INCLUDE_REGISTER
#define INCLUDE_REGISTER

#include <iostream.h>
#include <string.h>
#include "String.h"

#define REG_LOOP_START 5L
#define REG_LOOP_END 4L
#define REG_SUB 3L
#define REG_ADD 2L
#define REG_INDEXCOUNT 1L
#define REG_SPACE 0L

#define MINIMUM_ALLOCATE 100

class CRegister {
public:
    CRegister();
    CRegister(char *);

```

```

CRegister(long);
CRegister(const CRegister &);
~CRegister();
CRegister operator+(const CRegister &);
CRegister operator*(const CRegister &);
CRegister operator-(const CRegister &);
CRegister operator/(const CRegister &);
CRegister operator%(const CRegister &);
CRegister & operator++(int);
CRegister & operator--(int);
CRegister & operator+=(const CRegister &);
CRegister & operator+=(const long);
CRegister & operator-=(const CRegister &);
CRegister & operator-=(const long);
CRegister & operator*=(const CRegister &);
CRegister & operator/=(const CRegister &);
CRegister & operator%=(const CRegister &);
CRegister & operator*=(const long &);
CRegister & operator/=(const long &);
CRegister & operator%=(const long &);

operator long();
bool iszero();
char * toChar();
void operator=(const long);
void operator=(const CRegister &);

private:
char * m_pcValue;
long m_lCurrentSize;

void newSpace(long lNewSize);

CRegister & mull0();
CRegister & divl0();
CRegister & modl0();

void RemoveLeadingZero(char * &);
};

class CErrorMemory {
private:
char * m_pcError;
public:
CErrorMemory() {};
~CErrorMemory() {};
const char * Cause() const { return "MEMORY ERROR"; }
};

#endif

```

2. Laufzeitoptimierter Interpreter für Register-Maschinen-Programme

2.1. D05.cpp

```

#include <iostream.h>
#include <fstream.h>
#include <stdio.h>
#include "convert.h"
#include "interpret.h"

#define MAX_PROGRAM_LENGTH 1048576

void Status(bool bStatusCode, bool bStatusData, bool bStatusLogging, ofstream ofsLogStream, char
* Program, CRegister Register[], CRegister rCount) {

cout << "<" << rCount.toChar() << "/" << strlen(Program) << "> ";
if (bStatusLogging)
ofsLogStream << "<" << rCount.toChar() << "/" << strlen(Program) << "> ";

```

```

    if (bStatusCode) {
        cout << Program << " . ";
        if (bStatusLogging)
            ofsLogStream << Program << " . ";
    }
    if (bStatusData) {
        long lMax = 0;
        for(lMax=79;lMax>1;lMax--) {
            if (!Register[lMax].iszero())
                break;
        }
        for(long i=1; i<=lMax; i++) {
            cout << Register[i].toChar();
            if (bStatusLogging)
                ofsLogStream << Register[i].toChar();
            if (i != lMax) {
                cout << ",";
                if (bStatusLogging)
                    ofsLogStream << ",";
            }
        }
    }
    if ((bStatusData || bStatusCode) && bStatusLogging)
        ofsLogStream << endl;
    cout << endl;
}

void main() {
    CRegister * Register = new CRegister [80];
    ofstream ofsLog;
    bool bData = true;
    bool bCode = false;
    bool bRun = true;
    bool bLog = true;
    long m,l;
    char * Program = new char [MAX_PROGRAM_LENGTH];
    char x[200];
    CRegister r;
    FILE * f = fopen("c:\\code.rm","r");

    x[0] = 0;

    m = fread(Program,sizeof(char),MAX_PROGRAM_LENGTH,f); Program[m] = 0;

    fclose(f);

    ofsLog.open("c:\\testout.log",ios::out);

    cout << "Interpreter für die Register-Maschine (5)" << endl;
    cout << "*****" << endl;
    cout << endl;

    SplitProgram(Program, Register);

    m = 0;

    Status(true, true, true, ofsLog, Program, Register, (CRegister)0L);

    cout << "Programmstart..." << endl;
    cout << "Eingabe: <nnn[c][d]> mit nnn = Anzahl der Durchläufe ohne Unterbrechung" << endl;
    cout << "                [c] : 'c' = kein Befehlscode, 'C' = Befehlscode anzeigen" <<
endl;
    cout << "                [d] : 'd' = keine Daten, 'D' = Daten anzeigen" << endl;
    cout << endl;

    l = 0;

    while ((x[0] != 'q') && bRun) {
        if (m <= 0) {
            cout << ">";
            cin >> x;
            cout << endl;

            if (x[0] == 'q') exit(0);

            m = atol(x);

```

```

        if (strchr(x,'c') != NULL) bCode = false; else if (strchr(x,'C') != NULL) bCode =
true;
        if (strchr(x,'d') != NULL) bData = false; else if (strchr(x,'D') != NULL) bData =
true;
        if (strchr(x,'l') != NULL) bLog = false; else if (strchr(x,'L') != NULL) bLog =
true;
    }
    if (m > 0)
        m--;

    bRun = RExec(Program, Register, r);

    l++;
    if ((!bCode) && (!bData)) {
        if (l % 3000 == 0) {
            Status(false, true, true, ofsLog, Program, Register, r);
        }
    } else
        Status(bCode, bData, bLog, ofsLog, Program, Register, r);
}
cout << "*****" << endl;
Status(true, true, true, ofsLog, Program, Register, r);
cin >> x;
ofsLog.close();
}

```

2.2. Interpret.cpp

```

#include <string.h>
#include <stdlib.h>
#include "interpret.h"
#include "convert.h"
#include "..\D03\register.h"

void SplitProgram(char * pcProgram, CRegister * & Register) {
    char * pcTemp = strchr(pcProgram, PLAIN_PROG_END);

    // retrieve register contents
    if (pcTemp != NULL) {
        long i = 0;
        long j = 0;

        pcTemp[0] = 0;

        pcTemp++;

        do {
            i++;
            j = strchr(pcTemp, PLAIN_DATA_SPACE) - pcTemp;
            pcTemp[j] = 0;
            Register[i] = CRegister(pcTemp);
            pcTemp += j;
            pcTemp++;
        } while (strchr(pcTemp, PLAIN_DATA_SPACE) != NULL);
    }

    // remove comments
    pcTemp = strchr(pcProgram, PLAIN_COMMENT_START);
    while (pcTemp != NULL) {
        char * pcTemp2 = strchr(pcProgram, PLAIN_COMMENT_END);

        pcTemp2++;
        memmove(pcTemp, pcTemp2, strlen(pcTemp2)+1);
        pcTemp = strchr(pcProgram, PLAIN_COMMENT_START);
    }

    while (strchr(pcProgram, 10) != NULL) {
        pcTemp = strchr(pcProgram, 10);
        memmove(pcTemp, pcTemp+1, strlen(pcTemp)+1);
    }

    while (strchr(pcProgram, 13) != NULL) {
        pcTemp = strchr(pcProgram, 10);
        memmove(pcTemp, pcTemp+1, strlen(pcTemp)+1);
    }
}

```

```

    }
}

void RMLoop(char * Program, CRegister * & Register, CRegister & rCount) {
    char caShort[200];
    char caIndex[10];
    char * pcTemp = Program;
    long lDepth = 1;
    long lMaxDepth = 1;
    long lTemp = 0;
    long lIndex = 0;
    bool bComputed = false;

    pcTemp++;

    while (lDepth > 0) {
        char * pcTemp1 = strchr(pcTemp, PLAIN_LOOP_START);
        char * pcTemp2 = strchr(pcTemp, PLAIN_LOOP_END);
        char * pcTemp3 = __min(pcTemp1, pcTemp2);

        if (pcTemp3 == NULL)
            if (pcTemp1 == NULL)
                pcTemp3 = pcTemp2;
            else
                pcTemp3 = pcTemp1;

        if (pcTemp3 == pcTemp1) {
            lDepth++;
            pcTemp = pcTemp3;
            pcTemp++;
            if (lDepth > lMaxDepth) lMaxDepth = lDepth;
        } else {
            lDepth--;
            pcTemp = pcTemp3;
            pcTemp++;
        }
    }

    lTemp = strspn(pcTemp, "0123456789");

    lIndex = atol(pcTemp);

    if (lMaxDepth == 1) {
        ltoa(lIndex, caIndex, 10);

        strcpy(caShort, "(S");
        strcat(caShort, caIndex);
        strcat(caShort, " )");
        strcat(caShort, caIndex);

        if (strncmp(Program, caShort, strlen(caShort)) == 0) {
            Register[lIndex] = 0;
            bComputed = true;
        } else {
            char * pcTemp2 = strchr(Program, 'A');
            long lIndex2 = 0;
            char caIndex2[10];

            if (pcTemp2 != NULL) {
                pcTemp2++;
                lIndex2 = atol(pcTemp2);
                ltoa(lIndex2, caIndex2, 10);

                strcpy(caShort, "(S");
                strcat(caShort, caIndex);
                strcat(caShort, " A");
                strcat(caShort, caIndex2);
                strcat(caShort, " )");
                strcat(caShort, caIndex);

                if (strncmp(Program, caShort, strlen(caShort)) == 0) {
                    rCount += Register[lIndex];
                    rCount += Register[lIndex];
                    rCount += Register[lIndex];

                    Register[lIndex2] += Register[lIndex];
                }
            }
        }
    }
}

```

```

        Register[lIndex] = 0;
        bComputed = true;
    }
}

if (!bComputed) {
    char * pcTemp2 = strchr(Program, 'A');
    char * pcTemp3 = NULL;
    long lIndex2 = 0;
    long lIndex3 = 0;
    char caIndex2[10];
    char caIndex3[10];

    if (pcTemp2 != NULL) {
        pcTemp2++;
        pcTemp3 = strchr(pcTemp2, 'A');

        if (pcTemp3 != NULL) {
            pcTemp3++;

            lIndex2 = atol(pcTemp2);
            ltoa(lIndex2, caIndex2, 10);

            lIndex3 = atol(pcTemp3);
            ltoa(lIndex3, caIndex3, 10);

            strcpy(caShort, "(S");
            strcat(caShort, caIndex);
            strcat(caShort, " A");
            strcat(caShort, caIndex2);
            strcat(caShort, " A");
            strcat(caShort, caIndex3);
            strcat(caShort, " )");
            strcat(caShort, caIndex);

            if (strncmp(Program, caShort, strlen(caShort)) == 0) {
                rCount += Register[lIndex];
                rCount += Register[lIndex];
                rCount += Register[lIndex];
                rCount += Register[lIndex];

                Register[lIndex2] += Register[lIndex];
                Register[lIndex3] += Register[lIndex];
                Register[lIndex] = 0;
                bComputed = true;
            }
        }
    }
}

if (!bComputed) {
    char * pcTemp2 = strchr(Program, 'A');
    char * pcTemp3 = NULL;
    long lIndex2 = 0;
    long lIndex3 = 0;
    char caIndex2[10];
    char caIndex3[10];

    if (pcTemp2 != NULL) {
        pcTemp2++;
        pcTemp3 = strchr(pcTemp2, 'S');

        if (pcTemp3 != NULL) {
            pcTemp3++;

            lIndex2 = atol(pcTemp2);
            ltoa(lIndex2, caIndex2, 10);

            lIndex3 = atol(pcTemp3);
            ltoa(lIndex3, caIndex3, 10);

            strcpy(caShort, "(S");
            strcat(caShort, caIndex);
            strcat(caShort, " A");
            strcat(caShort, caIndex2);
            strcat(caShort, " S");
            strcat(caShort, caIndex3);
            strcat(caShort, " )");
            strcat(caShort, caIndex);
        }
    }
}

```



```

    l = atol(Program);

    Register[l]++;

    memmove(Program, Program+lTemp, strlen(Program) - lTemp + 1);

    rCount++;
}

void RMSub(char * Program, CRegister * & Register, CRegister & rCount) {
    long lTemp = strstr(Program, "0123456789");
    long l = 0;

    l = atol(Program);

    Register[l]--; // check if < 0 is implicit

    memmove(Program, Program+lTemp, strlen(Program) - lTemp + 1);

    rCount++;
}

bool RMExec(char * Program, CRegister * & Register, CRegister & rCount) {

    char * pcAdd = strchr(Program, PLAIN_ADD);
    char * pcSub = strchr(Program, PLAIN_SUB);
    char * pcLoop = strchr(Program, PLAIN_LOOP_START);
    char * pcTemp = NULL;

    if (pcAdd == NULL) pcAdd = (char*)0xFFFFFFFF;
    if (pcSub == NULL) pcSub = (char*)0xFFFFFFFF;
    if (pcLoop == NULL) pcLoop = (char*)0xFFFFFFFF;

    pcTemp = __min(pcAdd, __min(pcSub, pcLoop));

    if (pcTemp == (char*)0xFFFFFFFF)
        return false;
    else if (pcTemp == pcAdd) {
        pcAdd++;
        memmove(Program, pcAdd, strlen(pcAdd)+1);
        RMAdd(Program, Register, rCount);
    } else if (pcTemp == pcSub) {
        pcSub++;
        memmove(Program, pcSub, strlen(pcSub)+1);
        RMSub(Program, Register, rCount);
    } else if (pcTemp == pcLoop) {
        memmove(Program, pcLoop, strlen(pcLoop)+1);
        RMLoop(Program, Register, rCount);
    } else {
        return false;
    }
    return true;
}

```

2.3. Interpret.h

```

#ifndef INCLUDE_INTERPRET
#define INCLUDE_INTERPRET
#include "..\D03\register.h"
#include <stdlib.h>

void SplitProgram(char * pcProgram, CRegister * & Register);

bool RMExec(char * pcProgram, CRegister * & Register, CRegister & rCount);

#endif

```

2.4. Convert.h

```

#ifndef INCLUDE_CONVERT
#define INCLUDE_CONVERT

#include "..\D03\register.h"

```

```

#define PLAIN_LOOP_START '('
#define PLAIN_LOOP_END ')'
#define PLAIN_ADD 'A'
#define PLAIN_SUB 'S'
#define PLAIN_PROG_END '.'
#define PLAIN_DATA_SPACE ','
#define PLAIN_SPACE ' '
#define PLAIN_SPACE2 '\n'
#define PLAIN_SPACE3 '\r'
#define PLAIN_COMMENT_START '['
#define PLAIN_COMMENT_END ']'

#define PLAIN_S_LOOP_START "("
#define PLAIN_S_LOOP_END ")"
#define PLAIN_S_ADD "A"
#define PLAIN_S_SUB "S"
#define PLAIN_S_PROG_END "."
#define PLAIN_S_DATA_SPACE ","
#define PLAIN_S_SPACE " "

#endif

```

3. Interpreter für Register-Maschinen-Programme

3.1. D03.cpp

```

#include <iostream.h>
#include <fstream.h>
#include <stdio.h>
#include "convert.h"
#include "interpret.h"

void main() {
    CRegister r1,r2;
    CRegister ** pprRegister = NULL;
    ofstream ofsLog;
    bool bData = true;
    bool bCode = false;
    long l,m;
    char * pc;
    char x[99999];
    FILE * f = fopen("c:\\code.rm","r");

    l = fread(x,sizeof(char),99999,f); x[l] = 0;

    fclose(f);

    ofsLog.open("testout.log",ios::out);

    cout << "Interpreter für die Register-Maschine" << endl;
    cout << "*****" << endl;
    cout << "Gegebenes Programm:" << endl;
    cout << x << endl;
    plain2register(x,r1,r2);
    cout << endl;
    cout << "Interne Darstellung:" << endl;
    pc = r1.toChar();
    cout << pc;
    pc = r2.toChar();
    cout << " ." << pc << endl;

    cout << endl;

    pprRegister = new CRegister* [20];
    for(l=0;l<20;l++)
        pprRegister[l] = new CRegister;
    *(pprRegister[0]) = r1;
    *(pprRegister[1]) = r2;

    l = 0;
    m = 0;

    cout << "Programmstart..." << endl;

```

```

    cout << "Eingabe: <nnn[c][d]> mit nnn = Anzahl der Durchläufe ohne Unterbrechung" << endl;
    cout << "                [c] : 'c' = kein Befehlscode, 'C' = Befehlscode anzeigen" <<
endl;
    cout << "                [d] : 'd' = keine Daten, 'D' = Daten anzeigen" << endl;
    cout << endl;

    while ((x[0] != 'q') && (!pprRegister[0]->iszero())) {
        if (m <= 0) {
            cout << ">";
            cin >> x;
            cin.clear();
            m = atol(x);
            if (strchr(x,'c') != NULL)
                bCode = false;
            else if (strchr(x,'C') != NULL) bCode = true;
            if (strchr(x,'d') != NULL)
                bData = false;
            else if (strchr(x,'D') != NULL) bData = true;

            cout << endl;
        }
        if (m > 0)
            m--;
        l++;
        cout << "<" << l << "> ";
        RExec(pprRegister, 20);
        register2plain(pc,*pprRegister[0],*pprRegister[1],bCode,bData);
        cout << pc << endl;
        if (strlen(pc) > 3)
            ofsLog << "<" << l << "> " << pc << endl;
        delete [] pc;
    }
    cout << "*****" << endl;
    cin >> x;
    ofsLog.close();

    for(l=0;l<20;l++)
        delete pprRegister[l];
    delete [] pprRegister;
}

```

3.2. Convert.cpp

```

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <iostream.h>
#include <math.h>
#include "convert.h"

char * mirror(char * pcString) {
    long l = strlen(pcString);
    char * pcRev = new char [l+1];
    long i = 0;

    memset(pcRev, 0, l+1);

    for(i=0;i<l;i++)
        pcRev[l-i-1] = pcString[i];

    return pcRev;
}

bool register2plain(char * & pcProgram, CRegister rRegisterProgram, CRegister rRegisterData,
bool bConvertProgram = true, bool bConvertData = true) {
    CRegister rCheck;
    long index = 0;
    char buffer[20];
    bool bAllowNumber = false;
    char * pcTemp;

    pcTemp = new char [strlen(rRegisterProgram.toChar()) + strlen(rRegisterData.toChar()*2 + 1 +
20]; // 20 = buffer
    memset(pcTemp, 0, strlen(rRegisterProgram.toChar()) + strlen(rRegisterData.toChar()*2 + 1 +
20);
}

```

```

if (bConvertProgram) {
    while (!rRegisterProgram.iszero()) {
        rCheck = rRegisterProgram;
        rCheck %= 10;
        switch (atol(rCheck.toChar())) {
            case REG_ADD :
                if (bAllowNumber) {
                    ltoa(index,buffer,10);
                    strcat(pcTemp,buffer);
                }
                index = 0;
                strcat(pcTemp," ");
                strcat(pcTemp,PLAIN_S_ADD);
                bAllowNumber = true;
                break;
            case REG_SUB:
                if (bAllowNumber) {
                    ltoa(index,buffer,10);
                    strcat(pcTemp,buffer);
                }
                index = 0;
                strcat(pcTemp," ");
                strcat(pcTemp,PLAIN_S_SUB);
                bAllowNumber = true;
                break;
            case REG_LOOP_START:
                if (bAllowNumber) {
                    ltoa(index,buffer,10);
                    strcat(pcTemp,buffer);
                }
                index = 0;
                strcat(pcTemp," ");
                strcat(pcTemp,PLAIN_S_LOOP_START);
                bAllowNumber = false;
                break;
            case REG_LOOP_END:
                if (bAllowNumber) {
                    ltoa(index,buffer,10);
                    strcat(pcTemp,buffer);
                }
                index = 0;
                strcat(pcTemp," ");
                strcat(pcTemp,PLAIN_S_LOOP_END);
                bAllowNumber = true;
                break;
            case REG_INDEXCOUNT:
                index++;
            default:
                break;
        }
        rRegisterProgram /= 10;
    }
    if (bAllowNumber) {
        ltoa(index,buffer,10);
        strcat(pcTemp,buffer);
    }
}

strcat(pcTemp,PLAIN_S_SPACE);
strcat(pcTemp,PLAIN_S_PROG_END);
strcat(pcTemp,PLAIN_S_SPACE);

if (bConvertData) {
    index = 0;
    while (!rRegisterData.iszero()) {
        rCheck = rRegisterData;
        rCheck %= 10;
        switch (atol(rCheck.toChar())) {
            case REG_SPACE:
                ltoa(index,buffer,10);
                strcat(pcTemp,buffer);
                index = 0;
                strcat(pcTemp,PLAIN_S_DATA_SPACE);
                break;
            case REG_INDEXCOUNT:
                index++;
            default:
                break;
        }
    }
}

```

```

    }
    rRegisterData /= 10;
}
ltoa(index,buffer,10);
strcat(pcTemp,buffer);
}

pcProgram = new char [strlen(pcTemp) + 1];
memset(pcProgram, 0, strlen(pcTemp)+1);
strcpy(pcProgram,pcTemp);
delete [] pcTemp;

return true;
}

bool plain2register(char * pcProgram, CRegister & rRegisterProgram, CRegister & rRegisterData) {
char * pcCopy      = new char [strlen(pcProgram)+1];
char * pcCode      = NULL;
char * pcCodeR     = NULL;
char * pcData      = NULL;
char * pcDataR     = NULL;
long l             = 0;
long i             = 0;
long index         = 0;
CRegister index2;
long power         = 0;
bool               bComment = false;

memset(pcCopy, 0, strlen(pcProgram)+1);
strcpy(pcCopy, pcProgram);
pcData = strchr(pcCopy, PLAIN_PROG_END);
if (pcData != NULL) {
pcCode = new char [pcData - pcCopy + 1];
memset(pcCode, 0,pcData - pcCopy + 1);
strncpy(pcCode, pcCopy, pcData - pcCopy); // without separator

pcData = new char [strlen(pcCopy) - strlen(pcCode) + 1];
memset(pcData, 0, strlen(pcCopy) - strlen(pcCode) + 1);
strcpy(pcData, strchr(pcCopy, PLAIN_PROG_END) + 1);
}

// first step: turnaround
pcCodeR = mirror(pcCode);
pcDataR = mirror(pcData);

// second step: convert the program
l = strlen(pcCodeR);
rRegisterProgram = 0;
for(i=0;i<l;i++) {
if (i % 100 == 0) cout << i << endl;
if (bComment) {
if (pcCodeR[i] == PLAIN_COMMENT_START)
bComment = false;
} else {
switch (pcCodeR[i]) {
case PLAIN_COMMENT_END:
bComment = true;
break;
case PLAIN_ADD:
while (index > 0) { rRegisterProgram *= 10; rRegisterProgram++; index--; };
rRegisterProgram *= 10;
rRegisterProgram += REG_ADD;
index = 0; power = 0;
break;
case PLAIN_SUB:
while (index > 0) { rRegisterProgram *= 10; rRegisterProgram++; index--; };
rRegisterProgram *= 10;
rRegisterProgram += REG_SUB;
index = 0; power = 0;
break;
case PLAIN_LOOP_START:
rRegisterProgram *= 10;
rRegisterProgram += REG_LOOP_START;
index = 0; power = 0;
break;
case PLAIN_LOOP_END:
while (index > 0) { rRegisterProgram *= 10; rRegisterProgram++; index--; };
rRegisterProgram *= 10;
}
}
}
}

```

```

        rRegisterProgram += REG_LOOP_END;
        index = 0; power = 0;
        break;
    case PLAIN_SPACE:
    case PLAIN_SPACE2:
    case PLAIN_SPACE3:
        break;
    case '0':
    case '1':
    case '2':
    case '3':
    case '4':
    case '5':
    case '6':
    case '7':
    case '8':
    case '9':
        index = index + (long)(pcCodeR[i] - '0') * (long)pow(10,power);
        power++;
        break;
    default:
        cout << "\nSTOP" << endl;
        break;
    }
}
}

// third step: convert the data
// second step: convert the program
index2 = 0;
l = strlen(pcDataR);
rRegisterData = 0;
for(i=0;i<l;i++) {
    cout << i << endl;
    switch (pcDataR[i]) {
    case PLAIN_SPACE:
    case PLAIN_SPACE2:
    case PLAIN_SPACE3:
    case PLAIN_DATA_SPACE:
        rRegisterData *= 10;
        while (!index2.iszero()) { rRegisterData *= 10; rRegisterData++; index2--; };
        index2 = 0; power = 0;
        break;
    case '0':
    case '1':
    case '2':
    case '3':
    case '4':
    case '5':
    case '6':
    case '7':
    case '8':
    case '9':
        index2 += (long)(pcDataR[i] - '0') * (long)pow(10,power);
        power++;
        break;
    default:
        cout << "\nSTOP" << endl;
        break;
    }
}

if (pcCopy != NULL) delete [] pcCopy;
if (pcCode != NULL) delete [] pcCode;
if (pcCodeR != NULL) delete [] pcCodeR;
if (pcData != NULL) delete [] pcData;
if (pcDataR != NULL) delete [] pcDataR;

return true;
}

```

3.3. Convert.h

```

#ifndef INCLUDE_CONVERT
#define INCLUDE_CONVERT

```

```

#include "register.h"

#define PLAIN_LOOP_START '('
#define PLAIN_LOOP_END ')'
#define PLAIN_ADD 'A'
#define PLAIN_SUB 'S'
#define PLAIN_PROG_END '.'
#define PLAIN_DATA_SPACE ','
#define PLAIN_SPACE ' '
#define PLAIN_SPACE2 '\n'
#define PLAIN_SPACE3 '\r'
#define PLAIN_COMMENT_START '['
#define PLAIN_COMMENT_END ']'

#define PLAIN_S_LOOP_START "("
#define PLAIN_S_LOOP_END ")"
#define PLAIN_S_ADD "A"
#define PLAIN_S_SUB "S"
#define PLAIN_S_PROG_END "."
#define PLAIN_S_DATA_SPACE ","
#define PLAIN_S_SPACE " "

char * mirror(char * pcString);

bool plain2register(char * pcProgram, CRegister & prRegisterProgram, CRegister &
prRegisterData);
bool register2plain(char * & pcProgram, CRegister rRegisterProgram, CRegister rRegisterData,
bool bConvertProgram, bool bConvertData);

#endif

```

3.4. Interpret.cpp

```

#include "interpret.h"
#include "register.h"

void RMComputeCodeIndex(CRegister & rProgram, CRegister & rNextCode, CRegister & rIndex) {

    rNextCode = rProgram;
    rNextCode /= 10;
    while ((long)rNextCode == REG_INDEXCOUNT) {
        (rIndex)++;
        rProgram /= 10;
        rNextCode = rProgram;
        rNextCode /= 10;
    }
}

void RMComputeDataIndex(CRegister & rData, CRegister & rNextData, CRegister & rIndex, CRegister
& rIndexData) {

    rIndexData = 1;

    while (!rIndex.iszero()) {
        rNextData = rData;
        rNextData /= 10;
        if ((long)rNextData == REG_SPACE) rIndex--;
        rData /= 10;
        rIndexData *= 10;
    }
    rIndexData /= 10;
}

void RMComputeLoopIndex(CRegister & rProgram, CRegister & rNextCode, CRegister & rHelp1,
CRegister & rLoopDepth, CRegister & rIndex, CRegister & rLoopProg, CRegister & rProgShift,
CRegister & rLoopShift) {

    rHelp1 = rProgram;
    rHelp1 /= 10;
    rLoopDepth = 1;
    rProgShift = 1;

    while (!rLoopDepth.iszero()) {
        rNextCode = rHelp1;

```

```

    rNextCode %= 10;
    rHelp1 /= 10;
    switch (rNextCode) {
    case REG_LOOP_START:
        rLoopDepth++;
        break;
    case REG_LOOP_END:
        rLoopDepth--;
        break;
    case REG_ADD:
        break;
    case REG_SUB:
        break;
    case REG_INDEXCOUNT:
        break;
    default:
        cout << "STOP!!!" << endl;
        break;
    }
    rProgShift *= 10;
}

rLoopProg = rProgram;
rLoopProg %= rProgShift;
rLoopShift = rProgShift;

rProgShift /= 10; // remove loop end sign
rLoopProg /= 10;

// set NextCode to index of loop register
rNextCode = rHelp1;
rNextCode %= 10;
rHelp1 /= 10;
rIndex = 0;
while (rNextCode == 1) {
    rNextCode = rHelp1;
    rNextCode %= 10;
    rHelp1 /= 10;
    rIndex++;
}
}

void RMLoop(CRegister & rProgram, CRegister & rNextData, CRegister & rLoopProg, CRegister &
rProgShift, CRegister & rLoopShift) {
    if (!rNextData.iszero()) {
        rProgShift /= 10; // for iszero check
        while (!rProgShift.iszero()) {
            rProgShift /= 10;
            rProgram *= 10;
        }
        rProgram = rProgram + rLoopProg;
    } else {
        while (!rLoopShift.iszero()) {
            rLoopShift /= 10;
            rProgram /= 10;
        }
        rNextData = rProgram;
        rNextData %= 10;
        while (rNextData == 1) {
            rProgram /= 10;
            rNextData = rProgram;
            rNextData %= 10;
        }
    }
}

void RMTTest(CRegister & rIndex, CRegister & rData, CRegister & rNextData, CRegister & rHelp,
CRegister & rIndexData) {

    rHelp = rData;
    rIndexData = rData;
    rIndexData *= 10;
    rIndexData *= 10;

    while (!rIndex.iszero()) {
        rNextData = rHelp;
        rNextData %= 10;
    }
}

```

```

        if (rNextData.iszero()) rIndex--; // == REG_SPACE
        rHelp /= 10;
        rIndexData /= 10;
    }
    rNextData = rIndexData;
    rNextData %= 10;
}

void RMAAdd(CRegister & rIndex, CRegister & rData, CRegister & rNextData, CRegister & rHelp,
CRegister & rIndexData) {

    rHelp = rData;

    RMComputeDataIndex(rHelp, rNextData, rIndex, rIndexData);

    rHelp = rData;
    rHelp %= rIndexData;
    rData -= rHelp;
    rData *= 10;
    rData += rHelp;
    rData += rIndexData;
}

void RMSub(CRegister & rIndex, CRegister & rData, CRegister & rNextData, CRegister & rHelp,
CRegister & rIndexData) {

    rHelp = rData;

    RMComputeDataIndex(rHelp, rNextData, rIndex, rIndexData);

    rHelp = rIndexData;
    rHelp /= 10;
    rNextData = rData;
    rNextData %= rIndexData;
    rNextData /= rHelp;
    if (!rNextData.iszero()) {
        rHelp = rData;
        rHelp %= rIndexData;
        rData -= rHelp;
        rIndexData /= 10;
        rHelp -= rIndexData;
        rData /= 10;
        rData = rData + rHelp;
    }
}

bool RMExec(CRegister ** pprRegister, long lCount) {
    CRegister * prProgram = pprRegister[ 0];
    CRegister * prData     = pprRegister[ 1];
    CRegister * prNextCode = pprRegister[ 2];
    CRegister * prLoopDepth = pprRegister[ 3];
    CRegister * prIndex     = pprRegister[ 4];
    CRegister * prNextData = pprRegister[ 5];
    CRegister * prIndexData = pprRegister[ 6];
    CRegister * prLoopProg = pprRegister[ 7];
    CRegister * prProgShift = pprRegister[ 8];
    CRegister * prLoopShift = pprRegister[ 9];
    CRegister * prHelp1     = pprRegister[10];

    *prNextCode = *prProgram;
    *prNextCode %= 10;
    switch((long)(*prNextCode)) {
    case REG_ADD :
        *prProgram /= 10;
        RMComputeCodeIndex(*prProgram, *prNextCode, *prIndex);
        RMAAdd(*prIndex, *prData, *prNextData, *prHelp1, *prIndexData);
        break;
    case REG_SUB:
        *prProgram /= 10;
        RMComputeCodeIndex(*prProgram, *prNextCode, *prIndex);
        RMSub(*prIndex, *prData, *prNextData, *prHelp1, *prIndexData);
        break;
    case REG_LOOP_START:
        RMComputeLoopIndex(*prProgram, *prNextCode, *prHelp1, *prLoopDepth, *prIndex,
        *prLoopProg, *prProgShift, *prLoopShift);
        RMTTest(*prIndex, *prData, *prNextData, *prHelp1, *prIndexData);
        RMLoop(*prProgram, *prNextData, *prLoopProg, *prProgShift, *prLoopShift);
        break;
    }
}

```

```

    default:
        return false;
        break;
    }
    return true;
}

```

3.5. Interpret.h

```

#ifndef INCLUDE_INTERPRET
#define INCLUDE_INTERPRET
#include "register.h"
#include <stdlib.h>

bool RMExec(CRegister ** pprRegister, long lCount);

#endif

```

4. Compiler

4.1. Vollständige Übersicht der Klassen und Strukturen

CErrorCompiler	
Dient dem Compiler als Abbruchkriterium. Wird in der Klasse CParse in den verschiedenen Parse-Funktionen verwendet um einen Fehler anzuzeigen, der nicht behoben werden kann, und der zum Stopp des Compilierens führen soll.	
Methode/Eigenschaft	Beschreibung
CErrorCompiler	Konstruktor
~CErrorCompiler	Destruktor
Cause	liefert den Grund für den Abbruch. Der Grund ist z. Zt. fest auf "ERROR" gestellt.

CToken	
Liest von einem Eingabestrom (Datei) und kann über einen Puffer Token zurückliefern. Stellt eine Funktion zur Ausgabe in einen Ausgabestrom (Datei) und eine Logging-Funktion zur Verfügung. Wird von CParse benutzt.	
Methode/Eigenschaft	Beschreibung
CToken	Konstruktor
~CToken	Destruktor
OpenOutput	öffnet die Eingabedatei
OpenInput	öffnet die Ausgabedatei
OpenLog	öffnet die Log-Datei
Close	schließt alle offenen Dateien
NextToken	liefert das nächste Token
PutBack	schiebt die gegebene Zeichenkette zurück in den Puffer, diese Zeichenkette wird als nächstes aus dem Puffer zurückgeliefert
Write	schreibt in die Ausgabedatei
Log	schreibt in die Log-Datei
ifstream * m_pifsIn	Eingabedatei

CToken	
Liest von einem Eingabestrom (Datei) und kann über einen Puffer Token zurückliefern. Stellt eine Funktion zur Ausgabe in einen Ausgabestrom (Datei) und eine Logging-Funktion zur Verfügung. Wird von CParse benutzt.	
Methode/Eigenschaft	Beschreibung
<code>ofstream * m_pofsOut</code>	Ausgabedatei
<code>ofstream * m_pofsLog</code>	Log-Datei
<code>char m_caBuffer[MAX_BUFFER]</code>	Puffer der Eingabedatei

CParse	
Parser für C++-Code. Enthält zusätzlich zur reinen parse-Funktionalität noch den Code des Übersetzers.	
Methode/Eigenschaft	Beschreibung
<code>CParse</code>	Konstruktor
<code>~CParse</code>	Destruktor
<code>ParseProgram</code>	compiliert die im Konstruktor angegebene Datei
<code>CToken tParse;</code>	stellt Funktionen zum Zugriff auf den Eingabe-Token-Strom, den Ausgabestrom und Logdatei zur Verfügung
<code>SObject * psTop;</code>	Zeiger auf die doppelt verkettete Liste der Variablen und Funktionen
<code>SObject * psLastVariable;</code>	zuletzt verwendete Variable
<code>SObject * psLastFunction;</code>	zuletzt aufgerufene Funktion
<code>SObject * psCurrentFunction;</code>	aktuell zu bearbeitende Funktion
<code>SObject sLastValue;</code>	zuletzt verwendeter (Zahlen-)Wert
<code>eOPERATOR nLastOperator;</code>	zuletzt verwendeter Operator
<code>long lRegister;</code>	Anzahl der bisher verwendeten Register bzw. höchster verwendeter Register-Index
<code>AddObject</code>	fügt Variable, Funktion oder Wert in die SObject-Liste an der richtigen Stelle hinzu
<code>FindObject</code>	sucht einen Eintrag in der SObject-Liste
<code>FindOperator</code>	ermittelt, welcher Operator gegeben wurde
<code>AddCode</code>	fügt Code an die Funktion in der SObject-Liste an
<code>CodeFixup</code>	führt den Fixup im Code durch
<code>AddInitValues</code>	fügt die Initialisierung der Variablen hinzu
<code>ParseFunction</code>	bearbeitet Funktionsdefinitionen
<code>ParseFunctionType</code>	bearbeitet den Funktionstyp der Funktionsdefinition
<code>ParseNewFunctionName</code>	bearbeitet den Namen der Funktionsdefinition: trägt den neuen Namen in die SObject-Liste ein

CParser	
Parser für C++-Code. Enthält zusätzlich zur reinen parse-Funktionalität noch den Code des Übersetzers.	
Methode/Eigenschaft	Beschreibung
ParseParameterList	bearbeitet die (ev. leere) Parameterliste in der Funktionsdefinition
ParseFilledParameterList	bearbeitet eine nicht leere Parameterliste in der Funktionsdefinition
ParseDeclaration	bearbeitet eine einzelne Deklaration einer Übergabevariablen oder einer Deklaration einer lokalen Variablen
ParseInitValue	bearbeitet die Zuweisung eines Initialwertes
ParseVariableType	bearbeitet den Typ der Variable bei einer Deklaration
ParseNewVariable	bearbeitet den Namen einer Variable bei einer Deklaration: fügt den neuen Namen in die SObject-Liste ein
ParseBlock	bearbeitet den Funktionsrumpf
ParseDeclarationList	bearbeitet eine (ev. leere) Deklarationsliste für lokale Variablen innerhalb einer Funktion
ParseTermList	bearbeitet eine (ev. leere) Anweisungsliste innerhalb einer Funktion
ParseTerm	bearbeitet eine einzelne Anweisung (if/if-else/while/Zuweisung)
ParseIfTerm	bearbeitet eine if-Anweisung
ParseWhileTerm	bearbeitet eine While-Schleife
ParseCondition	bearbeitet die Bedingung in einer if-Anweisung oder einer While-Schleife
ParseNotExpression	bearbeitet die Negation eines Ausdrucks
ParseNotOperator	bearbeitet den Negationsoperator
ParseLogicalExpression	bearbeitet logische Ausdrücke
ParseKnownVariableNumber	bearbeitet bekannte Variablen oder Zahlen-Konstanten
ParseKnownVariable	bearbeitet bekannte Variablen
ParseAssignment	bearbeitet Zuweisungen
ParseFunctionCall	bearbeitet Funktionsaufrufe (durch Call-By-Copy implementiert)
ParseKnownFunctionName	bearbeitet bekannte Funktionsnamen
ParseCallParameterList	bearbeitet die (ev. leere) Übergabeliste der Parameter eines Funktionsaufrufs
ParseFilledCallParameterList	bearbeitet eine nichtleere Übergabeliste der Parameter eines Funktionsaufrufs
ParseCallParameter	bearbeitet einen einzelnen Übergabeparameter
ParsePostfixOperator	bearbeitet die Postfix-Operatoren (++ und --)

CParser	
Parser für C++-Code. Enthält zusätzlich zur reinen parse-Funktionalität noch den Code des Übersetzers.	
Methode/Eigenschaft	Beschreibung
ParseAssignmentOperator	bearbeitet die Zuweisungsoperatoren =, +=, -=, *=, /=, %=
ParseOperator	bearbeitet einen arithmetischen Operator +, -, *, /, %
ParseLogicalOperator	bearbeitet logische Operatoren (momentan nur == und !=)
ParseRelationOperator	bearbeitet relationale Operatoren (>, >=, <, <=)
ParseNumber	bearbeitet Zahlen-Konstanten
ParseExpected	bearbeitet erwartete Eingabezeichen
ParseSemicolon	bearbeitet „;“
ParseBlockStart	bearbeitet „{“
ParseBlockEnd	bearbeitet „}“
ParseListStart	bearbeitet „(“
ParseListEnd	bearbeitet „)“
ParseListSeparator	bearbeitet „,“ (Komma)
TestFunction	testet, ob nächster Token im Puffer eine Funktion sein könnte
TestDeclaration	testet, ob nächster Token im Puffer eine Deklaration sein könnte
TestInitValue	testet, ob nächster Token im Puffer ein Initialer Wert für eine Variable (lokal oder Funktionsparameter) sein könnte
TestTerm	testet, ob nächster Token im Puffer ein Term sein könnte
TestIfTerm	testet, ob nächster Token im Puffer eine If-Anweisung sein könnte
TestWhileTerm	testet, ob nächster Token im Puffer eine While-Schleife sein könnte
TestAssignment	testet, ob nächster Token im Puffer eine Zuweisung sein könnte
TestElseTerm	testet, ob nächster Token im Puffer eine Else-Anweisung sein könnte
TestKnownVariable	testet, ob nächster Token im Puffer eine bekannte Variable sein könnte
TestNotExpression	testet, ob nächster Token im Puffer eine Negation sein könnte
TestNotOperator	testet, ob nächster Token im Puffer das Negations-Symbol „!“ sein könnte
TestCondition	testet, ob nächster Token im Puffer eine Bedingung sein könnte

C_Parse	
Parser für C++-Code. Enthält zusätzlich zur reinen parse-Funktionalität noch den Code des Übersetzers.	
Methode/Eigenschaft	Beschreibung
TestRelationOperator	testet, ob nächster Token im Puffer eine Relation sein könnte
TestPostfixOperator	testet, ob nächster Token im Puffer eine Postfix-Operator sein könnte
TestOperator	testet, ob nächster Token im Puffer ein Operator (+,-,...) sein könnte
TestFunctionCall	testet, ob nächster Token im Puffer eine Funktionsaufruf sein könnte
TestFilledCallParameterList	testet, ob nächster Token im Puffer eine nichtleere Parameterliste eines Funktionsaufrufs sein könnte
TestCallParameter	testet, ob nächster Token im Puffer ein Übergabeparameter sein könnte
TestBlockEnd	testet, ob nächster Token im Puffer das Block-Ende Zeichen sein könnte
TestListEnd	testet, ob nächster Token im Puffer das Ende einer Liste anzeigen könnte
TestListSeparator	testet, ob nächster Token im Puffer das Listen-Trennzeichen sein könnte
TestKnownVariableNumber	testet, ob nächster Token im Puffer eine bekannte Variable oder eine Zahlenkonstante sein könnte
TestNumber	testet, ob nächster Token im Puffer eine Zahlen-Konstante sein könnte
WritePostfix	erstellt Code für postfix Rechenoperationen
WriteAssignment	erstellt Code für Rechenoperationen
WriteRelation	erstellt Code für Vergleiche

SObject	
Nimmt Informationen zu Variablen und Funktionen auf. Erzeugt eine mehrfach verkettete Liste.	
Methode/Eigenschaft	Beschreibung
char * pcName	Name der Variable oder Funktion
char * pcCode	Code zur Funktion
eOBJECT_TYPE eType	Typ des Listenelements
SObject * psNext	nächstes Listenelement
SObject * psLast	vorhergehendes Listenelement
SObject * psDown	untergeordnetes Listenelement
SObject * psUp	übergeordnetes Listenelement

SObject	
Nimmt Informationen zu Variablen und Funktionen auf. Erzeugt eine mehrfach verkettete Liste.	
Methode/Eigenschaft	Beschreibung
long lRegister	Index des verwendeten Registers für die Variable, die diesem Listenelement entspricht
long lValue	Wert der Zahlenkonstante
long lInitValue	Initialwert der Zahlenkonstante

4.2. D04.cpp

```
#include <iostream.h>
#include <stdlib.h>
#include <string.h>
#include "token.h"
#include "parse.h"

void main() {
    char caBuffer[200];
    CParse p("testin.cpp", "testout.txt", "testlog.txt");

    try {
        p.ParseProgram();
    }
    catch (CErrorCompiler err) {
        cout << endl;
        cout << err.Cause() << endl;
    }

    cout << "Program compiled." << endl;
    cin >> caBuffer;
}

```

4.3. CParse.cpp

```
#include <stdlib.h>
#include <string.h>
#include "token.h"
#include "parse.h"

// -----
CParse::CParse(char * pcInput, char * pcOutput, char * pcLog) {
    psTop = NULL;

    lRegister = REGISTER_COUNT;

    tParse.OpenInput(pcInput);
    tParse.OpenOutput(pcOutput);
    tParse.OpenLog(pcLog);

    sLastValue.eType = OBJECT_VALUE;
    sLastValue.lRegister = 0;
    sLastValue.pcCode = NULL;
    sLastValue.pcName = NULL;
    sLastValue.psDown = NULL;
    sLastValue.psLast = NULL;
    sLastValue.psNext = NULL;
    sLastValue.psUp = NULL;
}

CParse::~CParse() {
    SObject * psHeadNow = psTop;

    while (psHeadNow != NULL) {
        SObject * psObjectNow = psHeadNow->psDown;
        SObject * psDelete = NULL;
    }
}

```

```

        while (psObjectNow != NULL) {
            SObject * psDelete = psObjectNow;
            psObjectNow = psObjectNow->psNext;
            if (psDelete->pcCode != NULL)
                delete [] psDelete->pcCode;
            if (psDelete->pcName != NULL)
                delete [] psDelete->pcName;
            delete psDelete;
        }
        psDelete = psHeadNow;
        psHeadNow = psHeadNow->psNext;
        if (psDelete->pcCode != NULL)
            delete [] psDelete->pcCode;
        if (psDelete->pcName != NULL)
            delete [] psDelete->pcName;
        delete psDelete;
    }

    tParse.Close();
}

SObject * CParse::AddObject(eOBJECT_TYPE eType, const char * pcName) {
    SObject * psNew = new SObject;

    psNew->eType = eType;
    psNew->psNext = NULL;
    psNew->psLast = psNew;
    psNew->pcCode = NULL;

    if (eType == OBJECT_FUNCTION) {
        psNew->pcName = new char [strlen(pcName)+1];
        strcpy(psNew->pcName, pcName);
        psNew->lRegister = lRegister; // just remember next register...
        psNew->psNext = psTop;
        if (psTop != NULL)
            psTop->psLast = psNew;
        psTop = psNew;
        psNew->psUp = NULL;
        psNew->psDown = NULL;
        psCurrentFunction = psNew;
    } else if (eType == OBJECT_VARIABLE) {
        psNew->pcName = new char [strlen(pcName)+1];
        strcpy(psNew->pcName, pcName);
        psNew->lInitValue = 0;
        psNew->lRegister = lRegister;
        lRegister++;
        psNew->psNext = psTop->psDown;
        if (psTop->psDown != NULL)
            psTop->psDown->psLast = psNew;
        psTop->psDown = psNew;
        psNew->psUp = psTop;
        psTop->psDown = psNew;
        psLastVariable = psNew;
    }

    return psNew;
}

SObject * CParse::FindObject(const char * pcName) {
    SObject * psHeadNow = psTop;

    while (psHeadNow != NULL) {
        if (strcmp(psHeadNow->pcName, pcName) == 0) {
            psLastFunction = psHeadNow;
            return psHeadNow;
        } else {
            SObject * psObjectNow = psHeadNow->psDown;
            while (psObjectNow != NULL) {
                if (strcmp(psObjectNow->pcName, pcName) == 0) {
                    psLastVariable = psObjectNow;
                    return psObjectNow;
                } else
                    psObjectNow = psObjectNow->psNext;
            }
            psHeadNow = psHeadNow->psNext;
        }
    }
}

```

```

    return NULL;
}

eOPERATOR CParse::FindOperator(const char * pcOperator) {
    long l = 0;

    for(l=0;l<OPERATOR_COUNT;l++) {
        if (strcmp(pcOperator,caOPERATOR[l]) == 0)
            return (eOPERATOR)l;
    }

    return OPERATOR_NONE;
}

void CParse::AddCode(char * & pcCode, const char * pcAddCode) {
    if (pcAddCode != NULL) {
        if (pcCode == NULL) {
            pcCode = new char [strlen(pcAddCode)+1];
            strcpy(pcCode, pcAddCode);
        } else {
            char * pcTemp = new char [strlen(pcCode)+strlen(pcAddCode)+1];
            strcpy(pcTemp, pcCode);
            strcat(pcTemp, pcAddCode);
            delete [] pcCode;
            pcCode = pcTemp;
        }
    }
}

void CParse::AddCode(char * & pcCode, const char * pcAddCode1, const char * pcAddCode2) {
    AddCode(pcCode, pcAddCode1);
    AddCode(pcCode, pcAddCode2);
}

void CParse::AddCode(char * & pcCode, const char * pcAddCode1, const char * pcAddCode2, const
char * pcAddCode3) {
    AddCode(pcCode,pcAddCode1, pcAddCode2);
    AddCode(pcCode, pcAddCode3);
}

void CParse::AddCode(char * & pcCode, const char * pcAddCode1, const char * pcAddCode2, const
char * pcAddCode3, const char * pcAddCode4) {
    AddCode(pcCode,pcAddCode1, pcAddCode2,pcAddCode3);
    AddCode(pcCode, pcAddCode4);
}

void CParse::AddCode(char * & pcCode, long lCode) {
    char buffer[10];
    ltoa(lCode, buffer, 10);
    AddCode(pcCode, buffer);
}

void CParse::CodeFixup(char * & pcCode, const char * pcSubst, long lRegister) {
    char * pcTemp = new char [strlen(pcCode)+1];
    char caBuffer[10];

    strcpy(pcTemp, pcCode);
    ltoa(lRegister, caBuffer, 10);

    while (strstr(pcTemp, pcSubst) != NULL) {
        char * pcTemp2 = new char [strlen(pcTemp) + strlen(caBuffer) - strlen(pcSubst) + 1];
        memset(pcTemp2, 0, strlen(pcTemp) + strlen(caBuffer) - strlen(pcSubst) + 1);
        if (strstr(pcTemp, pcSubst) != pcTemp)
            strncpy(pcTemp2, pcTemp, strstr(pcTemp, pcSubst) - pcTemp);

        strcat(pcTemp2, caBuffer);

        strcat(pcTemp2, strstr(pcTemp, pcSubst) + strlen(pcSubst));

        delete [] pcTemp;

        pcTemp = pcTemp2;
        pcTemp2 = NULL;
    }

    delete [] pcCode;
    pcCode = pcTemp;
}

```

```

void CParse::CodeFixup(char * & pcCode, long lRegister) {
    CodeFixup(pcCode, "*1*", lRegister);
}

void CParse::CodeFixup(char * & pcCode, long lRegister1, long lRegister2) {
    CodeFixup(pcCode, "*1*", lRegister1);
    CodeFixup(pcCode, "*2*", lRegister2);
}

void CParse::CodeFixup(char * & pcCode, long lRegister1, long lRegister2, long lRegister3) {
    CodeFixup(pcCode, "*1*", lRegister1);
    CodeFixup(pcCode, "*2*", lRegister2);
    CodeFixup(pcCode, "*3*", lRegister3);
}

void CParse::AddInitValues(char * & pcCode) {
    long ** pplValue= NULL;
    long lMaxIndex    = 0;
    long l            = 0;
    SObject * psHeadNow = psTop;

    while (psHeadNow != NULL) {
        SObject * psObjectNow = psHeadNow->psDown;
        while (psObjectNow != NULL) {
            if (psObjectNow->lRegister > lMaxIndex)
                lMaxIndex = psObjectNow->lRegister;
            psObjectNow = psObjectNow->psNext;
        }
        psHeadNow = psHeadNow->psNext;
    }

    if (lMaxIndex > 0) {
        pplValue = new long * [lMaxIndex+1];
        for(l=0;l<=lMaxIndex;l++) {
            pplValue[l] = NULL;
            psHeadNow = psTop;
            while (psHeadNow != NULL) {
                SObject * psObjectNow = psHeadNow->psDown;
                while (psObjectNow != NULL) {
                    if (psObjectNow->lRegister == l) {
                        pplValue[l] = &(psObjectNow->lInitValue);
                        break;
                    }
                    psObjectNow = psObjectNow->psNext;
                }
                if (pplValue[l] != NULL)
                    break;
                psHeadNow = psHeadNow->psNext;
            }
        }

        for(l=1;l<=lMaxIndex;l++) {
            if (pplValue[l] != NULL)
                AddCode(pcCode, *(pplValue[l]));
            else
                AddCode(pcCode, 0L);
            AddCode(pcCode, ",");
        }
        AddCode(pcCode, "0");
    }
}

void CParse::ParseProgram() {
    char caBuffer[MAX_TOKEN];
    SObject * psProgram;
    tParse.Log("program");

    tParse >> caBuffer;
    while (strlen(caBuffer) > 0) {
        tParse.PutBack(caBuffer);
        ParseFunction();
        tParse >> caBuffer;
    }
}

```

```

tParse.Log("program end");

psProgram = FindObject("main");

AddCode(psProgram->pcCode, RMCODE_INT_INIT);
AddCode(psProgram->pcCode, RMCODE_EOP);

AddInitValues(psProgram->pcCode);

tParse << psProgram->pcCode;
}

void CParse::ParseFunction() {
    tParse.Log("function");

    ParseFunctionType();

    ParseNewFunctionName();

    ParseListStart();

    ParseParameterList();

    ParseListEnd();

    ParseBlock();
}

void CParse::ParseFunctionType() {
    tParse.Log("function type");

    ParseExpected("void");
}

void CParse::ParseNewFunctionName() {
    char caBuffer[MAX_TOKEN];
    tParse.Log("new function name");

    tParse >> caBuffer;

    AddObject(OBJECT_FUNCTION, caBuffer);
}

void CParse::ParseParameterList() {
    tParse.Log("parameter list");

    if (!TestListEnd())
        ParseFilledParameterList();
}

void CParse::ParseFilledParameterList() {
    tParse.Log("filled parameter list");

    ParseDeclaration();

    if (TestListSeparator()) {
        ParseListSeparator();
        ParseFilledParameterList();
    }
}

void CParse::ParseDeclaration() {
    tParse.Log("declaration");

    ParseVariableType();

    ParseNewVariable();

    if (TestInitValue()) {
        ParseInitValue();
    }
}

void CParse::ParseInitValue() {
    char caBuffer[MAX_TOKEN];
    long l = 0;
}

```

```

    tParse.Log("initial value");

    ParseExpected("=");

    tParse >> caBuffer;

    l = atol(caBuffer);

    psLastVariable->lInitValue = l;
}

void CParse::ParseVariableType() {
    tParse.Log("variable type");

    ParseExpected("CRegister");
}

void CParse::ParseNewVariable() {
    char caBuffer[MAX_TOKEN];
    tParse.Log("new variable");

    tParse >> caBuffer;

    AddObject(OBJECT_VARIABLE, caBuffer);

    AddCode(psCurrentFunction->pcCode, "[new variable: ", psLastVariable->pcName, " = R*1*]\n");
    CodeFixup(psCurrentFunction->pcCode, psLastVariable->lRegister);
}

void CParse::ParseBlock() {
    tParse.Log("block");

    ParseBlockStart();

    ParseDeclarationList();

    ParseTermList();

    ParseBlockEnd();
}

void CParse::ParseDeclarationList() {
    tParse.Log("declaration list");

    if (TestDeclaration()) {
        ParseDeclaration();
        ParseSemicolon();
        ParseDeclarationList();
    }
}

void CParse::ParseTermList() {
    tParse.Log("term list");

    if (TestTerm()) {
        ParseTerm();
        ParseTermList();
    }
}

void CParse::ParseTerm() {
    tParse.Log("term");

    if (TestIfTerm())
        ParseIfTerm();
    else if (TestWhileTerm())
        ParseWhileTerm();
    else if (TestAssignment())
        ParseAssignment();
    else {
        tParse.Log("Error: term, assignment or function call expected");
        throw CErrorCompiler();
    }
}

void CParse::ParseIfTerm() {
    char * pcPreCode= NULL;
    char * pcPostCode = NULL;

```

```

tParse.Log("if term");

ParseExpected("if");

AddCode(psCurrentFunction->pcCode, "[if] ");

ParseCondition(REGISTER_CONDITION, pcPreCode, pcPostCode);

AddCode(psCurrentFunction->pcCode, pcPreCode);
AddCode(psCurrentFunction->pcCode, RMCODE_PRE_IF);

ParseBlock();

AddCode(psCurrentFunction->pcCode, RMCODE_POST_IF);
AddCode(psCurrentFunction->pcCode, pcPostCode);

if (TestElseTerm()) {
    if (pcPreCode != NULL) {
        delete [] pcPreCode;
        pcPreCode = NULL;
    }
    if (pcPostCode != NULL) {
        delete [] pcPostCode;
        pcPostCode = NULL;
    }

    ParseExpected("else");

    AddCode(psCurrentFunction->pcCode, pcPreCode);
    AddCode(psCurrentFunction->pcCode, RMCODE_PRE_ELSE);

    ParseBlock();

    AddCode(psCurrentFunction->pcCode, RMCODE_POST_ELSE);
    AddCode(psCurrentFunction->pcCode, pcPostCode);
}

}

void CParse::ParseWhileTerm() {
    char * pcPreCode= NULL;
    char * pcPostCode = NULL;
    tParse.Log("while term");

    ParseExpected("while");

    AddCode(psCurrentFunction->pcCode, "[while] ");

    ParseCondition(REGISTER_CONDITION, pcPreCode, pcPostCode);

    AddCode(psCurrentFunction->pcCode, RMCODE_PREPRE_WHILE);
    AddCode(psCurrentFunction->pcCode, pcPreCode);
    AddCode(psCurrentFunction->pcCode, RMCODE_PRE_WHILE);

    ParseBlock();

    AddCode(psCurrentFunction->pcCode, RMCODE_POST_WHILE);
    AddCode(psCurrentFunction->pcCode, pcPostCode);
}

void CParse::ParseCondition(long lRegister, char * & pcPreCode, char * & pcPostCode) {
    tParse.Log("condition");

    ParseListStart();

    ParseLogicalExpression(lRegister, pcPreCode, pcPostCode);

    ParseListEnd();
}

void CParse::ParseLogicalExpression(long lRegister, char * & pcPreCode, char * & pcPostCode) {
    tParse.Log("logical expression");

    if (TestNotOperator()) {
        ParseNotOperator();
        ParseCondition(lRegister, pcPreCode, pcPostCode);
    } else if (TestCondition()) {

```

```

        ParseCondition(lRegister, pcPreCode, pcPostCode);
        ParseLogicalOperator();
        ParseCondition(lRegister, pcPreCode, pcPostCode);
    } else if (TestKnownVariableNumber()) {
        SObject * psLeftVariableNumber = NULL;
        SObject * psRightVariableNumber = NULL;
        eOPERATOR nRelationOperator = OPERATOR_NONE;

        ParseKnownVariableNumber();
        psLeftVariableNumber = psLastVariable;

        ParseRelationOperator();
        nRelationOperator = nLastOperator;

        ParseKnownVariableNumber();
        psRightVariableNumber = psLastVariable;

        WriteRelation(pcPreCode, lRegister, psLeftVariableNumber, nRelationOperator,
psRightVariableNumber);

    } else {
        tParse.Log("Error: not operator, condition, known variable or number expected");
        throw CErrorCompiler();
    }
}

void CParse::ParseNotOperator() {
    char caBuffer[MAX_TOKEN];
    tParse.Log("not operator");

    tParse >> caBuffer;
}

void CParse::ParseKnownVariableNumber() {
    tParse.Log("known variable or number");

    if (TestKnownVariable())
        ParseKnownVariable();
    else if (TestNumber())
        ParseNumber();
    else {
        tParse.Log("Error: known variable or number expected");
        throw CErrorCompiler();
    }
}

void CParse::ParseKnownVariable() {
    char caBuffer[MAX_TOKEN];
    SObject * psObject = NULL;
    tParse.Log("known variable");

    tParse >> caBuffer;

    psObject = FindObject(caBuffer);

    if (psObject != NULL) {
        if (psObject->eType != OBJECT_VARIABLE) {
            tParse.Log("Error: known _variable_ expected");
            throw CErrorCompiler();
        }
    } else {
        tParse.Log("Error: _known_ variable expected");
        throw CErrorCompiler();
    }
}

void CParse::ParseAssignment() {
    tParse.Log("assignment");

    if (TestKnownVariable()) {
        ParseKnownVariable();
        if (TestPostfixOperator()) {
            ParsePostfixOperator();
            WritePostfix(psCurrentFunction->pcCode, psLastVariable, nLastOperator);
            ParseSemicolon();
        } else {
            SObject * psLeftObject = psLastVariable;
            SObject * psMiddleObject = NULL;

```

```

    eOPERATOR nLeftOperator = OPERATOR_NONE;

    ParseAssignmentOperator();
    ParseKnownVariableNumber();

    psMiddleObject = psLastVariable;
    nLeftOperator = nLastOperator;

    if (TestOperator()) {
        SObject * psRightObject = NULL;
        eOPERATOR nRightOperator = OPERATOR_NONE;

        ParseOperator();
        ParseKnownVariableNumber();

        nRightOperator = nLastOperator;
        psRightObject = psLastVariable;

        WriteAssignment(psCurrentFunction->pcCode, psLeftObject, nLeftOperator,
psMiddleObject, nRightOperator, psRightObject);
    } else {
        WriteAssignment(psCurrentFunction->pcCode, psLeftObject, nLeftOperator,
psMiddleObject);
    }
    ParseSemicolon();
}
}
AddCode(psCurrentFunction->pcCode, "\n");
} else if (TestFunctionCall()) {
    ParseFunctionCall();
    AddCode(psCurrentFunction->pcCode, "\n");
    ParseSemicolon();
} else {
    tParse.Log("Error: known variable or function call expected");
    throw CErrorCompiler();
}
}
}

void CParse::ParseFunctionCall() {
    char * pcPreCall = NULL;
    char * pcPostCall = NULL;

    tParse.Log("function call");

    ParseKnownFunctionName();

    ParseListStart();

    ParseCallParameterList(pcPreCall, pcPostCall);

    ParseListEnd();

    AddCode(psCurrentFunction->pcCode, "[call to \"];
    AddCode(psCurrentFunction->pcCode, psLastFunction->pcName);
    AddCode(psCurrentFunction->pcCode, "\"];");

    AddCode(psCurrentFunction->pcCode, pcPreCall);

    AddCode(psCurrentFunction->pcCode, "[function body] ");
    AddCode(psCurrentFunction->pcCode, psLastFunction->pcCode);
    AddCode(psCurrentFunction->pcCode, pcPostCall);
}

void CParse::ParseKnownFunctionName() {
    char caBuffer[MAX_TOKEN];
    SObject * psObject = NULL;
    tParse.Log("known function name");

    tParse >> caBuffer;

    psObject = FindObject(caBuffer);

    if (psObject != NULL) {
        if (psObject->eType != OBJECT_FUNCTION) {
            tParse.Log("Error: unknown _function_ name");
            throw CErrorCompiler();
        }
    }
} else {
    tParse.Log("Error: unknown function _name_");
}
}

```

```

        throw CErrorCompiler();
    }
}

void CParse::ParseCallParameterList(char * & pcPreCall, char * & pcPostCall) {
    tParse.Log("call parameter list");

    if (TestFilledCallParameterList())
        ParseFilledCallParameterList(0, pcPreCall, pcPostCall);
}

void CParse::ParseFilledCallParameterList(long lParamNumber, char * & pcPreCall, char * &
pcPostCall) {
    tParse.Log("filled call parameter list");

    ParseCallParameter(lParamNumber, pcPreCall, pcPostCall);
    if (TestListSeparator()) {
        ParseListSeparator();
        ParseFilledCallParameterList(lParamNumber+1, pcPreCall, pcPostCall);
    }
}

void CParse::ParseCallParameter(long lParamNumber, char * & pcPreCall, char * & pcPostCall) {
    tParse.Log("call parameter");

    ParseKnownVariableNumber();

    // cleanup destination register
    AddCode(pcPreCall, RMCODE_INIT);
    CodeFixup(pcPreCall, psLastFunction->lRegister + lParamNumber);

    if (psLastVariable->eType == OBJECT_VALUE) {
        long l = 0;
        AddCode(pcPreCall, "[copy parameter: register *1* is *2*] ");
        CodeFixup(pcPreCall, psLastFunction->lRegister + lParamNumber, psLastVariable->lValue);
        AddCode(pcPreCall, RMCODE_INIT);
        for(l=0;l<psLastVariable->lValue;l++) {
            AddCode(pcPreCall, RMCODE_ADD);
            CodeFixup(pcPreCall, psLastFunction->lRegister + lParamNumber);
        }
    } else {
        AddCode(pcPreCall, "[copy parameter: function register *1*, caller register *2*] ");
        CodeFixup(pcPreCall, psLastFunction->lRegister + lParamNumber, psLastVariable->lRegister);

        // copy source to destination register (empty source register)
        AddCode(pcPreCall, RMCODE_INIT);
        CodeFixup(pcPreCall, psLastFunction->lRegister + lParamNumber);
        AddCode(pcPreCall, RMCODE_ADDDESTROY);
        CodeFixup(pcPreCall, psLastVariable->lRegister, psLastFunction->lRegister + lParamNumber);

        AddCode(pcPostCall, "[copy parameter: backward] ");
        // copy destination to source register (empty destination register)
        AddCode(pcPostCall, RMCODE_ADDDESTROY);
        CodeFixup(pcPostCall, psLastFunction->lRegister + lParamNumber, psLastVariable-
>lRegister);
    }
}

void CParse::ParsePostfixOperator() {
    char caBuffer[MAX_TOKEN];
    tParse.Log("postfix operator");

    tParse >> caBuffer;

    nLastOperator = FindOperator(caBuffer);
    if (nLastOperator == OPERATOR_NONE) {
        tParse.Log("Error: no operator");
        throw CErrorCompiler();
    }
}

void CParse::ParseAssignmentOperator() {
    char caBuffer[MAX_TOKEN];
    tParse.Log("assignment operator");

    tParse >> caBuffer;

    nLastOperator = FindOperator(caBuffer);
}

```

```

        if (nLastOperator == OPERATOR_NONE) {
            tParse.Log("Error: no operator");
            throw CErrorCompiler();
        }
    }

void CParse::ParseOperator() {
    char caBuffer[MAX_TOKEN];
    tParse.Log("operator");

    tParse >> caBuffer;

    nLastOperator = FindOperator(caBuffer);

    if (nLastOperator == OPERATOR_NONE) {
        tParse.Log("Error: no operator");
        throw CErrorCompiler();
    }
}

void CParse::ParseLogicalOperator() {
    char caBuffer[MAX_TOKEN];
    tParse.Log("logical-operator");

    tParse >> caBuffer;

    nLastOperator = FindOperator(caBuffer);

    if (nLastOperator == OPERATOR_NONE) {
        tParse.Log("Error: no operator");
        throw CErrorCompiler();
    }
}

void CParse::ParseRelationOperator() {
    char caBuffer[MAX_TOKEN];
    tParse.Log("relation operator");

    tParse >> caBuffer;

    nLastOperator = FindOperator(caBuffer);

    if (nLastOperator == OPERATOR_NONE) {
        tParse.Log("Error: no operator");
        throw CErrorCompiler();
    }
}

void CParse::ParseNumber() {
    char caBuffer[MAX_TOKEN];
    tParse.Log("number");

    tParse >> caBuffer;

    sLastValue.lValue = atol(caBuffer);

    psLastVariable = & sLastValue;
}

void CParse::ParseSemicolon() {
    char caBuffer[MAX_TOKEN];
    tParse.Log("semicolon ';'");

    tParse >> caBuffer;

    if (strcmp(caBuffer, ";") != 0) {
        tParse.Log("Error: no ';' found");
        throw CErrorCompiler();
    }
}

void CParse::ParseListStart() {
    char caBuffer[MAX_TOKEN];
    tParse.Log("list start '('");

    tParse >> caBuffer;

    if (strcmp(caBuffer, "(") != 0) {
        tParse.Log("Error: no '(' found");
    }
}

```

```

        throw CErrorCompiler();
    }
}

void CParse::ParseListEnd() {
    char caBuffer[MAX_TOKEN];
    tParse.Log("list end \\\"");

    tParse >> caBuffer;
    if (strcmp(caBuffer, ")") != 0) {
        tParse.Log("Error: no ')' found");
        throw CErrorCompiler();
    }
}

void CParse::ParseBlockStart() {
    char caBuffer[MAX_TOKEN];
    tParse.Log("block start \\{");

    tParse >> caBuffer;
    if (strcmp(caBuffer, "{") != 0) {
        tParse.Log("Error: no '{' found");
        throw CErrorCompiler();
    }
}

void CParse::ParseBlockEnd() {
    char caBuffer[MAX_TOKEN];
    tParse.Log("block end \\}");

    tParse >> caBuffer;
    if (strcmp(caBuffer, ")") != 0) {
        tParse.Log("Error: no ')' found");
        throw CErrorCompiler();
    }
}

void CParse::ParseListSeparator() {
    char caBuffer[MAX_TOKEN];
    tParse.Log("list separator \\\",");

    tParse >> caBuffer;
    if (strcmp(caBuffer, ",") != 0) {
        tParse.Log("Error: no ',' found");
        throw CErrorCompiler();
    }
}

void CParse::ParseExpected(char * pcExpected) {
    char caBuffer[MAX_TOKEN];
    tParse.Log("expected \"%s\"", pcExpected);

    tParse >> caBuffer;
    if (strcmp(caBuffer, pcExpected) != 0) {
        tParse.Log("Error: no '%s' found", caBuffer);
        throw CErrorCompiler();
    }
}

// -----
bool CParse::TestDeclaration() {
    char caBuffer[MAX_TOKEN];
    bool bTest = false;
    tParse.Log("test variable declaration");

    tParse >> caBuffer;
    tParse.PutBack(caBuffer);

    if (strcmp(caBuffer, "CRegister") == 0)
        bTest = true;

    return bTest;
}

bool CParse::TestInitValue() {
    char caBuffer[MAX_TOKEN];
    bool bTest = false;
    tParse.Log("test initial value");
}

```

```

    tParse >> caBuffer;
    tParse.PutBack(caBuffer);

    if (strcmp(caBuffer, "=") == 0)
        bTest = true;

    return bTest;
}

bool CParse::TestTerm() {
    tParse.Log("test term");
    return (TestIfTerm() || TestWhileTerm() || TestAssignment());
}

bool CParse::TestIfTerm() {
    char caBuffer[MAX_TOKEN];
    bool bTest = false;
    tParse.Log("test if term");

    tParse >> caBuffer;
    tParse.PutBack(caBuffer);

    if (strcmp(caBuffer, "if") == 0)
        bTest = true;

    return bTest;
}

bool CParse::TestWhileTerm() {
    char caBuffer[MAX_TOKEN];
    bool bTest = false;
    tParse.Log("test while term");

    tParse >> caBuffer;
    tParse.PutBack(caBuffer);

    if (strcmp(caBuffer, "while") == 0)
        bTest = true;

    return bTest;
}

bool CParse::TestAssignment() {
    tParse.Log("test assignment");
    return (TestKnownVariable() | TestFunctionCall());
}

bool CParse::TestElseTerm() {
    char caBuffer[MAX_TOKEN];
    bool bTest = false;
    tParse.Log("test test else term");

    tParse >> caBuffer;
    tParse.PutBack(caBuffer);

    if (strcmp(caBuffer, "else") == 0)
        bTest = true;

    return bTest;
}

bool CParse::TestKnownVariable() {
    char caBuffer[MAX_TOKEN];
    bool bTest = false;
    SObject * psObject = NULL;
    tParse.Log("test known variable");

    tParse >> caBuffer;
    tParse.PutBack(caBuffer);

    psObject = FindObject(caBuffer);

    if (psObject != NULL)
        if (psObject->eType == OBJECT_VARIABLE)
            bTest = true;

    return bTest;
}

```

```

}

bool CParse::TestNotOperator() {
    char caBuffer[MAX_TOKEN];
    bool bTest = false;
    tParse.Log("test not operator");

    tParse >> caBuffer;
    tParse.PutBack(caBuffer);

    if (strcmp(caBuffer,"!") == 0)
        bTest = true;

    return bTest;
}

bool CParse::TestCondition() {
    char caBuffer[MAX_TOKEN];
    bool bTest = false;
    tParse.Log("test condition");

    tParse >> caBuffer;
    tParse.PutBack(caBuffer);

    if (strcmp(caBuffer,"(") == 0)
        bTest = true;

    return bTest;
}

bool CParse::TestRelationOperator() {
    char caBuffer[MAX_TOKEN];
    bool bTest = false;
    tParse.Log("test relation");

    tParse >> caBuffer;
    tParse.PutBack(caBuffer);

    if ((strcmp(caBuffer,"<") == 0) ||
        (strcmp(caBuffer,"<=") == 0) ||
        (strcmp(caBuffer,">") == 0) ||
        (strcmp(caBuffer,">=") == 0) ||
        (strcmp(caBuffer,"==") == 0))
        bTest = true;

    return bTest;
}

bool CParse::TestPostfixOperator() {
    char caBuffer[MAX_TOKEN];
    bool bTest = false;
    tParse.Log("test postfix operator");

    tParse >> caBuffer;
    tParse.PutBack(caBuffer);

    if ((strcmp(caBuffer,"++") == 0) || (strcmp(caBuffer,"--") == 0))
        bTest = true;

    return bTest;
}

bool CParse::TestOperator() {
    char caBuffer[MAX_TOKEN];
    bool bTest = false;
    tParse.Log("test operator");

    tParse >> caBuffer;
    tParse.PutBack(caBuffer);

    if ((strcmp(caBuffer,"+") == 0) ||
        (strcmp(caBuffer,"-") == 0) ||
        (strcmp(caBuffer,"*") == 0) ||
        (strcmp(caBuffer,"/") == 0) ||
        (strcmp(caBuffer,"%") == 0))
        bTest = true;

    return bTest;
}

```

```

}

bool CParse::TestFunctionCall() {
    char caBuffer[MAX_TOKEN];
    SObject * psObject = NULL;
    bool bTest = false;
    tParse.Log("test function call");

    tParse >> caBuffer;
    tParse.PutBack(caBuffer);

    psObject = FindObject(caBuffer);

    if (psObject != NULL)
        if (psObject->eType == OBJECT_FUNCTION)
            bTest = true;

    return bTest;
}

bool CParse::TestFilledCallParameterList() {
    tParse.Log("test filled call parameter list");
    return TestKnownVariableNumber();
}

bool CParse::TestBlockEnd() {
    char caBuffer[MAX_TOKEN];
    bool bTest = false;
    tParse.Log("test block end");

    tParse >> caBuffer;
    tParse.PutBack(caBuffer);

    return bTest;
}

bool CParse::TestListEnd() {
    char caBuffer[MAX_TOKEN];
    bool bTest = false;
    tParse.Log("test list end");

    tParse >> caBuffer;
    tParse.PutBack(caBuffer);

    if (strcmp(caBuffer,")") == 0)
        bTest = true;

    return bTest;
}

bool CParse::TestKnownVariableNumber() {
    tParse.Log("test known variable or number");
    return (TestKnownVariable() | TestNumber());
}

bool CParse::TestNumber() {
    char caBuffer[MAX_TOKEN];
    long l = 0;
    char caNumber[MAX_TOKEN];
    bool bTest = false;
    tParse.Log("test number");

    tParse >> caBuffer;
    tParse.PutBack(caBuffer);

    l = atol(caBuffer);

    ltoa(l,caNumber, 10);

    if (strcmp(caNumber, caBuffer) == 0)
        bTest = true;

    return bTest;
}

bool CParse::TestListSeparator() {
    char caBuffer[MAX_TOKEN];

```

```

bool bTest = false;
tParse.Log("test list separator");

tParse >> caBuffer;
tParse.PutBack(caBuffer);

if (strcmp(caBuffer, ",") == 0)
    bTest = true;

return bTest;
}

void CParse::WritePostfix(char * & pcCode, SObject * psVariable, eOPERATOR nOperator) {
    AddCode(pcCode, "[\"];
    AddCode(pcCode, psVariable->pcName);
    AddCode(pcCode, (char*)caOPERATOR[nOperator]);
    AddCode(pcCode, "\"];

    switch (nOperator) {
    case OPERATOR_PLUSPLUS:
        AddCode(pcCode, RMCODE_ADD);
        CodeFixup(pcCode, psVariable->lRegister);
        break;
    case OPERATOR_MINUSMINUS:
        AddCode(pcCode, RMCODE_SUB);
        CodeFixup(pcCode, psVariable->lRegister);
        break;
    default:
        tParse.Log("ERROR: unknown postfix operator found");
        throw CErrorCompiler();
        break;
    }
}

void CParse::WriteAssignment(char * & pcCode, SObject * psLeftObject, long nLeftOperator,
SObject * psMiddleObject, long nRightOperator, SObject * psRightObject) {
    switch (nLeftOperator) {
    case OPERATOR_ASSIGN:
        switch (nRightOperator) {
        case OPERATOR_PLUS:
            tParse.Log("ERROR: NYI: OPERATOR_PLUS");
            throw CErrorCompiler();
            break;
        case OPERATOR_MINUS:
            tParse.Log("ERROR: NYI: OPERATOR_MINUS");
            throw CErrorCompiler();
            break;
        case OPERATOR_DIVIDE:
            tParse.Log("ERROR: NYI: OPERATOR_DIVIDE");
            throw CErrorCompiler();
            break;
        case OPERATOR_MULTIPLY:
            // a = b * c : while (b!) { b--; a = a + c; };
            tParse.Log("ERROR: NYI: OPERATOR_MULTIPLY");
            throw CErrorCompiler();
            break;
        case OPERATOR_MODULO:
            tParse.Log("ERROR: NYI: OPERATOR_MODULO");
            throw CErrorCompiler();
            break;
        default:
            tParse.Log("ERROR: unknown operator found");
            throw CErrorCompiler();
            break;
        }
    }
    break;
default:
    tParse.Log("ERROR: unsupported triple-operator on left side");
    throw CErrorCompiler();
    break;
}
}

void CParse::WriteAssignment(char * & pcCode, SObject * psLeftObject, long nLeftOperator,
SObject * psMiddleObject) {
    AddCode(pcCode, "[\"];
    AddCode(pcCode, psLeftObject->pcName);
    AddCode(pcCode, (char*)caOPERATOR[nLeftOperator]);
}

```

```

if (psMiddleObject->eType == OBJECT_VALUE)
    AddCode(pcCode, psMiddleObject->lValue);
else
    AddCode(pcCode, psMiddleObject->pcName);
AddCode(pcCode, "\\ " ");

switch (nLeftOperator) {
case OPERATOR_ASSIGN:
    if (psMiddleObject->eType == OBJECT_VARIABLE) {
        AddCode(pcCode, RMCODE_COPY);
        CodeFixup(pcCode, psMiddleObject->lRegister, psLeftObject->lRegister);
    } else if (psMiddleObject->eType == OBJECT_VALUE) {
        long l = 0;
        AddCode(pcCode, RMCODE_INIT);
        for(l=0;l<psMiddleObject->lValue;l++)
            AddCode(pcCode, RMCODE_ADD);
        CodeFixup(pcCode, psLeftObject->lRegister);
    }
    break;
case OPERATOR_PLUSASSIGN:
    if (psMiddleObject->eType == OBJECT_VARIABLE) {
        AddCode(pcCode, RMCODE_ADDCOPY);
        CodeFixup(pcCode, psMiddleObject->lRegister, psLeftObject->lRegister);
    } else if (psMiddleObject->eType == OBJECT_VALUE) {
        long l = 0;
        for(l=0;l<psMiddleObject->lValue;l++)
            AddCode(pcCode, RMCODE_ADD);
        CodeFixup(pcCode, psLeftObject->lRegister);
    }
    break;
case OPERATOR_MINUSASSIGN:
    if (psMiddleObject->eType == OBJECT_VARIABLE) {
        AddCode(pcCode, RMCODE_SUBCOPY);
        CodeFixup(pcCode, psMiddleObject->lRegister, psLeftObject->lRegister);
    } else if (psMiddleObject->eType == OBJECT_VALUE) {
        long l = 0;
        for(l=0;l<psMiddleObject->lValue;l++)
            AddCode(pcCode, RMCODE_SUB);
        CodeFixup(pcCode, psLeftObject->lRegister);
    }
    break;
case OPERATOR_DIVIDEASSIGN:
    if (psMiddleObject->eType == OBJECT_VARIABLE) {
        AddCode(pcCode, RMCODE_DIVASN_VAR);
        CodeFixup(pcCode, psLeftObject->lRegister, psMiddleObject->lRegister);
    } else if (psMiddleObject->eType == OBJECT_VALUE) {
        long l = 0;
        AddCode(pcCode, RMCODE_INIT);
        for(l=0;l<psMiddleObject->lValue;l++)
            AddCode(pcCode, RMCODE_ADD);
        CodeFixup(pcCode, REGISTER_VALUE);

        AddCode(pcCode, RMCODE_DIVASN_VAR);
        CodeFixup(pcCode, psLeftObject->lRegister, REGISTER_VALUE);
    }
    break;
case OPERATOR_MULTIPLYASSIGN:
    // a = b * c : while (b!) { b--; a = a + c; };

    if (psMiddleObject->eType == OBJECT_VARIABLE) {
        AddCode(pcCode, RMCODE_MULASN_VAR);
        CodeFixup(pcCode, psLeftObject->lRegister, psMiddleObject->lRegister);
    } else if (psMiddleObject->eType == OBJECT_VALUE) {
        long l = 0;
        AddCode(pcCode, RMCODE_INIT);
        for(l=0;l<psMiddleObject->lValue;l++)
            AddCode(pcCode, RMCODE_ADD);
        CodeFixup(pcCode, REGISTER_VALUE);

        AddCode(pcCode, RMCODE_MULASN_VAR);
        CodeFixup(pcCode, psLeftObject->lRegister, REGISTER_VALUE);
    }
    break;
case OPERATOR_MODULOASSIGN:
    if (psMiddleObject->eType == OBJECT_VARIABLE) {
        AddCode(pcCode, RMCODE_MODASN_VAR);
        CodeFixup(pcCode, psLeftObject->lRegister, psMiddleObject->lRegister);
    }

```

```

    } else if (psMiddleObject->eType == OBJECT_VALUE) {
        long l = 0;
        AddCode(pcCode, RMCODE_INIT);
        for(l=0;l<psMiddleObject->lValue;l++)
            AddCode(pcCode, RMCODE_ADD);
        CodeFixup(pcCode, REGISTER_VALUE);

        AddCode(pcCode, RMCODE_MODASN_VAR);
        CodeFixup(pcCode, psLeftObject->lRegister, REGISTER_VALUE);
    }
    break;
default:
    tParse.Log("ERROR: unsupported dual-operator on left side");
    throw CErrorCompiler();
    break;
}
}

void CParse::WriteRelation(char * & pcCode, long lRegister, SObject * psLeftVariableNumber,
eOPERATOR nRelationOperator, SObject * psRightVariableNumber) {
    switch(nRelationOperator) {
    case OPERATOR_CHECKGREATER:
        if (psRightVariableNumber->eType == OBJECT_VALUE) {
            if (psRightVariableNumber->lValue == 0) {
                AddCode(pcCode, RMCODE_CHECKNOTZERO);
                CodeFixup(pcCode, psLeftVariableNumber->lRegister);
            } else {
                tParse.Log("ERROR: NYI: OPERATOR_CHECKGREATER");
                throw CErrorCompiler();
            }
        } else if (psRightVariableNumber->eType == OBJECT_VARIABLE) {
            tParse.Log("ERROR: NYI: OPERATOR_CHECKGREATER");
            throw CErrorCompiler();
        }
        break;
    case OPERATOR_CHECKLESS:
        tParse.Log("ERROR: NYI: OPERATOR_CHECKLESS");
        throw CErrorCompiler();
        break;
    case OPERATOR_CHECKGREATEREQUAL:
        tParse.Log("ERROR: NYI: OPERATOR_CHECKGREATEREQUAL");
        throw CErrorCompiler();
        break;
    case OPERATOR_CHECKLESSEQUAL:
        tParse.Log("ERROR: NYI: OPERATOR_CHECKLESSEQUAL");
        throw CErrorCompiler();
        break;
    case OPERATOR_CHECKEQUAL:
        if (psRightVariableNumber->eType == OBJECT_VALUE) {
            long l = 0;
            AddCode(pcCode, "[", psLeftVariableNumber->pcName, " == R*1* "); CodeFixup(pcCode,
psRightVariableNumber->lValue);
            AddCode(pcCode, RMCODE_INIT);
            for(l=0;l<psRightVariableNumber->lValue;l++)
                AddCode(pcCode, RMCODE_ADD);
            CodeFixup(pcCode, REGISTER_VALUE);
            AddCode(pcCode, RMCODE_CHECKEQUAL);
            CodeFixup(pcCode, psLeftVariableNumber->lRegister, REGISTER_VALUE);
        } else if (psRightVariableNumber->eType == OBJECT_VARIABLE) {
            AddCode(pcCode, "[", psLeftVariableNumber->pcName, " == "); AddCode(pcCode,
psRightVariableNumber->pcName, "] ");
            AddCode(pcCode, RMCODE_CHECKEQUAL);
            CodeFixup(pcCode, psLeftVariableNumber->lRegister, psRightVariableNumber-
>lRegister);
        }
        break;
    case OPERATOR_CHECKNOTEQUAL:
        if (psRightVariableNumber->eType == OBJECT_VALUE) {
            long l = 0;
            AddCode(pcCode, "[", psLeftVariableNumber->pcName, " != R*1* "); CodeFixup(pcCode,
psRightVariableNumber->lValue);
            AddCode(pcCode, RMCODE_INIT);
            for(l=0;l<psRightVariableNumber->lValue;l++)
                AddCode(pcCode, RMCODE_ADD);
            CodeFixup(pcCode, REGISTER_VALUE);
            AddCode(pcCode, RMCODE_CHECKNOTEQUAL);
            CodeFixup(pcCode, psLeftVariableNumber->lRegister, REGISTER_VALUE);
        } else if (psRightVariableNumber->eType == OBJECT_VARIABLE) {

```

```

        AddCode(pcCode, "[", psLeftVariableNumber->pcName, " != "); AddCode(pcCode,
psRightVariableNumber->pcName, "]" );
        AddCode(pcCode, RMCODE_CHECKNOTEQUAL);
        CodeFixup(pcCode, psLeftVariableNumber->lRegister, psRightVariableNumber-
>lRegister);
    }
    break;
default:
    tParse.Log("ERROR: unsupported relation operator");
    throw CErrorCompiler();
    break;
}
AddCode(pcCode, "\n");
}
}

```

4.4. CParse.h

```

#ifndef INCLUDE_PARSE
#define INCLUDE_PARSE

#include <stdlib.h>
#include "token.h"

typedef enum neOBJECT_TYPE {
    OBJECT_UNKNOWN = 0,
    OBJECT_VARIABLE = 1,
    OBJECT_FUNCTION = 2,
    OBJECT_VALUE = 3
} eOBJECT_TYPE;

typedef enum neOPERATOR_TYPE {
    OPERATOR_UNKNOWN= 0,
    OPERATOR_UNARY = 1,
    OPERATOR_BINARY = 2,
    OPERATOR_FUNCTION = 3
} eOPERATOR_TYPE;

typedef enum neTERM_TYPE {
    TERMTYPE_NUMBER = 0,
    TERMTYPE_IDENTIFIER = 1
} eTERM_TYPE;

typedef enum neOPERATOR {
    OPERATOR_NONE = 0,
    OPERATOR_PLUSPLUS = 1,
    OPERATOR_MINUSMINUS = 2,
    OPERATOR_PLUS = 3,
    OPERATOR_MINUS = 4,
    OPERATOR_MULTIPLY = 5,
    OPERATOR_DIVIDE = 6,
    OPERATOR_MODULO = 7,
    OPERATOR_PLUSASSIGN = 8,
    OPERATOR_MINUSASSIGN = 9,
    OPERATOR_MULTIPLYASSIGN = 10,
    OPERATOR_DIVIDEASSIGN = 11,
    OPERATOR_MODULOASSIGN = 12,
    OPERATOR_ASSIGN = 13,
    OPERATOR_CHECKEQUAL = 14,
    OPERATOR_CHECKNOT = 15,
    OPERATOR_CHECKGREATER = 16,
    OPERATOR_CHECKLESS = 17,
    OPERATOR_CHECKGREATEREQUAL = 18,
    OPERATOR_CHECKLESSEQUAL = 19,
    OPERATOR_CHECKNOTEQUAL = 20
} eOPERATOR;

typedef enum neREGISTER {
    REGISTER_INVALID= -1,
    REGISTER_ELSE = 1,
    REGISTER_COPY = 2,
    REGISTER_HELP1 = 3,
    REGISTER_HELP2 = 4,
    REGISTER_WHILE = 5,
    REGISTER_MOD = 6,
    REGISTER_VALUE = 7,
    REGISTER_CONDITION = 9
} eREGISTER;

```

```

} eREGISTER;

const OPERATOR_COUNT      = 21;
const REGISTER_COUNT      = 10;

const char caOPERATOR[][3] = { "\0\0",
    "+", "--",
    "+\0", "-\0", "*\0", "/\0", "%\0",
    "+=", "-=", "*=", "/=", "%=",
    "=", "!=", "! \0",
    ">\0", "<\0", ">=", "<=", "!=" };

const char RMCODE_EOP[]      = " . ";
const char RMCODE_ADD[]     = "A*1* ";
const char RMCODE_SUB[]     = "S*1* ";
const char RMCODE_LOOPSTART[] = "(";
const char RMCODE_LOOPEND[] = ")*1* ";

const char RMCODE_ADDDESTROY[] = "(S*1* A*2* )*1* ";

const char RMCODE_COPY[]    = "(S*2* )*2* (S2 )2 (S*1* A*2* A2 )*1* (S2 A*1* )2 ";

const char RMCODE_ADDCOPY[] = "(S2 )2 (S*1* A*2* A2 )*1* (S2 A*1* )2 ";
const char RMCODE_SUBCOPY[] = "(S2 )2 (S*1* S*2* A2 )*1* (S2 A*1* )2 ";

const char RMCODE_MULASN_VAR[] = "(S2 )2 (S3 )3 (S4)4 [a nach R2 kopieren] (S*1* A2 )*1* [b nach R3 kopieren] (S*2* A3 A4 )*2* (S4 A*2* )4 [mul] (S3 [a+=a] (S2 A*1* A4 )2 (S4 A2 )4 )3 (S2 )2";

const char RMCODE_DIVASN_VAR[] = "(S2 )2 (S3 )3 (S4)4 [a nach R2 schieben] (S*1* A2 )*1* [b nach R3 sichern] (S*2* A3 A4 )*2* (S4 A*2* )4 [div] S3 (S3 S2 A4 )3 (S4 A3 )4 A3 ( (S3 S2 A4 )3 (S4 A3 )4 A*1* )2 (S3 )3"; // problem: schleife über R2 wird durchlaufen, auch wenn nur noch b-1 drin ist! d.h. 30/8 = 4 !!

const char RMCODE_MODASN_VAR[] = "(S2 )2 (S3 )3 (S4 )4 (S6 )6 [a nach R2 kopieren] (S*1* A2 A3 )*1* (S3 A*1* )3 [b nach R3 sichern] (S*2* A3 A4 )*2* (S4 A*2* )4 [div] S3 (S3 S2 A4 )3 (S4 A3 )4 A3 ( (S3 S2 A4 )3 (S4 A3 )4 A6 )2 (S3 )3 [ende-div] [R6 nach R2 kopieren] (S6 A2 A3 )6 (S3 A6 )3 [b nach R3 kopieren] (S*2* A3 A4 )*2* (S4 A*2* )4 [mul] S3 (S3 [a+=a] (S2 A6 A4 )2 (S4 A2 )4 )3 (S2 )2 (S6 S*1* )6 ";

const char RMCODE_INIT[]      = "(S*1* )*1* ";

const char RMCODE_INT_INIT[]  = "[internal cleanup] (S1 )1 (S2 )2 (S3 )3 (S4 )4 (S5 )5 (S6 )6 (S7 )7 (S9 )9 ";

const char RMCODE_PRE_IF[]    = "(S1 )1 A1 ( ";
const char RMCODE_POST_IF[]   = "(S1 )1 (S9 )9 )9 ";
const char RMCODE_PRE_ELSE[]  = "( ";
const char RMCODE_POST_ELSE[] = "(S1 )1 )1 ";

const char RMCODE_PREPRE_WHILE[] = "A5 ( (S5 )5 ";
const char RMCODE_PRE_WHILE[]   = "( ";
const char RMCODE_POST_WHILE[]  = "A5 (S9 )9 )9 )5 ";

const char RMCODE_CHECKEQUAL[] = "[R9,3,4 = 0] (S9 )9 (S3 )3 (S4 )4 [R3 = R*1*] (S*1* A3 A4 )*1* (S4 A*1* )4 [R9 -= R*2*] (S*2* S3 A4 )*2* (S4 A*2* )4 [R9 = R3] (S3 A9 )3 [R3 = R*2*] (S*2* A3 A4 )*2* (S4 A*2* )4 [R3 -= R*1*] (S*1* S3 A4 )*1* (S4 A*1* )4 [R9 += R3] (S3 A9 )3 [now: R9 = 0, if R*1* == R*2* !] A3 (S9 S3 )9 ((S3 )3 A9 )3 [now: R9 = 1, if R*1* == R*2* !] ";
const char RMCODE_CHECKNOTEQUAL[] = "[R9,3,4 = 0] (S9 )9 (S3 )3 (S4 )4 [R3 = R*1*] (S*1* A3 A4 )*1* (S4 A*1* )4 [R9 -= R*2*] (S*2* S3 A4 )*2* (S4 A*2* )4 [R9 = R3] (S3 A9 )3 [R3 = R*2*] (S*2* A3 A4 )*2* (S4 A*2* )4 [R3 -= R*1*] (S*1* S3 A4 )*1* (S4 A*1* )4 [R9 += R3] (S3 A9 )3 [now: R9 = 0, if R*1* == R*2* !] ";
const char RMCODE_CHECKNOT[]   = "[R3 = R*1*] (S*1* A9 A3 )*1* (S3 A*1* )3 [R9 = R*1*] A3 (S9 S3 )9 ((S3 )3 A9 )3 [now: R9 = 1, if R*1* == 0 and R9 = 0, if R*1* != 0 !] ";

const char RMCODE_CHECKNOTZERO[] = "(S2 )2 (S9 )9 (S*1* A2 A9 )*1* (S2 A*1* )2 ";

struct SObject {
    char *      pcName;
    char *      pcCode;
    eOBJECT_TYPE eType;
    SObject *   psNext;
    SObject *   psLast;
    SObject *   psDown;
    SObject *   psUp;
    long        lRegister;
    long        lValue;
    long        lInitValue;
};

```

```

};

class CParse {
public:
    CParse(char * pcInput, char * pcOutput, char * pcLog);
    ~CParse();
    void ParseProgram();
private:
    CToken    tParse;
    SObject   * psTop;
    SObject   * psLastVariable;
    SObject   * psLastFunction;
    SObject   * psCurrentFunction;
    SObject   sLastValue;
    eOPERATOR nLastOperator;
    long      lRegister;

    SObject * AddObject(eOBJECT_TYPE eType, const char * pcName);
    SObject * FindObject(const char * pcName);
    eOPERATOR FindOperator(const char * pcOperator);

    void AddCode(char * & pcCode, const char * pcAddCode);
    void AddCode(char * & pcCode, const char * pcAddCode1, const char * pcAddCode2);
    void AddCode(char * & pcCode, const char * pcAddCode1, const char * pcAddCode2, const char *
pcAddCode3);
    void AddCode(char * & pcCode, const char * pcAddCode1, const char * pcAddCode2, const char *
pcAddCode3, const char * pcAddCode4);
    void AddCode(char * & pcCode, long lCode);

    void CodeFixup(char * & pcCode, const char * pcSubst, long lRegister);
    void CodeFixup(char * & pcCode, long lRegister);
    void CodeFixup(char * & pcCode, long lRegister1, long lRegister2);
    void CodeFixup(char * & pcCode, long lRegister1, long lRegister2, long lRegister3);

    void AddInitValues(char * & pcCode);

    void ParseFunction();
    void ParseFunctionType();
    void ParseNewFunctionName();
    void ParseParameterList();
    void ParseFilledParameterList();
    void ParseDeclaration();
    void ParseInitValue();
    void ParseVariableType();
    void ParseNewVariable();
    void ParseBlock();
    void ParseDeclarationList();
    void ParseTermList();
    void ParseTerm();
    void ParseIfTerm();
    void ParseWhileTerm();
    void ParseCondition(long lRegister, char * & pcPreCode, char * & pcPostCode);
    void ParseNotExpression();
    void ParseNotOperator();
    void ParseLogicalExpression(long lRegister, char * & pcPreCode, char * & pcPostCode);
    void ParseKnownVariableNumber();
    void ParseKnownVariable();
    void ParseAssignment();
    void ParseFunctionCall();
    void ParseKnownFunctionName();
    void ParseCallParameterList(char * & pcPreCall, char * & pcPostCall);
    void ParseFilledCallParameterList(long lParamNumber, char * & pcPreCall, char * &
pcPostCall);
    void ParseCallParameter(long lParamNumber, char * & pcPreCall, char * & pcPostCall);
    void ParsePostfixOperator();
    void ParseAssignmentOperator();
    void ParseOperator();
    void ParseLogicalOperator();
    void ParseRelationOperator();
    void ParseNumber();

    void ParseExpected(char * pcExpeted);
    void ParseSemicolon();
    void ParseBlockStart();
    void ParseBlockEnd();
    void ParseListStart();
    void ParseListEnd();
    void ParseListSeparator();

```

```

bool TestFunction();
bool TestDeclaration();
bool TestInitValue();
bool TestTerm();
bool TestIfTerm();
bool TestWhileTerm();
bool TestAssignment();
bool TestElseTerm();
bool TestKnownVariable();
bool TestNotExpression();
bool TestNotOperator();
bool TestCondition();
bool TestRelationOperator();
bool TestPostfixOperator();
bool TestOperator();
bool TestFunctionCall();
bool TestFilledCallParameterList();
bool TestCallParameter();
bool TestBlockEnd();
bool TestListEnd();
bool TestListSeparator();
bool TestKnownVariableNumber();
bool TestNumber();

void WritePostfix(char * & pcCode, SObject * psVariable, eOPERATOR nOperator);
void WriteAssignment(char * & pcCode, SObject * psLeftObject, long nLeftOperator, SObject *
psMiddleObject, long nRightOperator, SObject * psRightObject);
void WriteAssignment(char * & pcCode, SObject * psLeftObject, long nLeftOperator, SObject *
psMiddleObject);
void WriteRelation(char * & pcCode, long lRegister, SObject * psLeftVariableNumber, eOPERATOR
nRelationOperator, SObject * psRightVariableNumber);
};

class CErrorCompiler {
private:
    char * m_pcError;
public:
    CErrorCompiler() {};
    ~CErrorCompiler() {};
    const char * Cause() const { return "ERROR"; }
};
#endif

```

4.5. CToken.cpp

```

#include <fstream.h>
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <stdarg.h>
#include "token.h"

CToken::CToken() {
    m_pifsIn = NULL;
    m_pofsOut = NULL;
    m_pofsLog = NULL;
}

CToken::CToken(char * pcInput, char * pcOutput, char * pcLog) {
    m_pifsIn = NULL;
    m_pofsOut = NULL;
    m_pofsLog = NULL;
    OpenInput(pcInput);
    OpenOutput(pcOutput);
    OpenLog(pcLog);
}

CToken::~CToken() {
    Close();
}

void CToken::OpenOutput(char * pcOutputFile) {
    if (m_pofsOut != NULL) {

```

```

        m_pofsOut->close();
        delete m_pofsOut;
    }
    m_pofsOut = new ofstream;
    m_pofsOut->open(pcOutputFile, ios::out, filebuf::openprot);
}

void CToken::OpenLog(char * pcLogFile) {
    if (m_pofsLog != NULL) {
        m_pofsLog->close();
        delete m_pofsLog;
    }
    m_pofsLog = new ofstream;
    m_pofsLog->open(pcLogFile, ios::out, filebuf::openprot);
}

void CToken::OpenInput(char * pcInputFile) {
    if (m_pifsIn != NULL) {
        m_pifsIn->close();
        delete m_pifsIn;
    }
    m_pifsIn = new ifstream;
    m_pifsIn->open(pcInputFile, ios::in, filebuf::sh_none);
    memset(m_caBuffer, 0, MAX_BUFFER);
}

void CToken::Close() {
    if (m_pifsIn != NULL) {
        m_pifsIn->close();
        delete m_pifsIn;
        m_pifsIn = NULL;
    }
    if (m_pofsOut != NULL) {
        m_pofsOut->close();
        delete m_pofsOut;
        m_pofsOut = NULL;
    }
}

void CToken::Log(char * pcLog) {
    *m_pofsLog << pcLog << endl;
    cout << pcLog << endl;
}

void CToken::Log(char * pcLog, char * pcLog2) {
    char caBuffer[MAX_BUFFER];
    sprintf(caBuffer, pcLog, pcLog2);
    *m_pofsLog << caBuffer << endl;
    cout << caBuffer << endl;
}

void CToken::Log(char * pcLog, long lLog) {
    char caBuffer[MAX_BUFFER];
    sprintf(caBuffer, pcLog, lLog);
    *m_pofsLog << caBuffer << endl;
    cout << caBuffer << endl;
}

char * CToken::NextToken() {
    char * pcReturn = NULL;
    char caBuffer[MAX_BUFFER];
    char caReturn[MAX_BUFFER];

    memset(caBuffer, 0, MAX_BUFFER);
    memset(caReturn, 0, MAX_BUFFER);

    if (strlen(m_caBuffer) <= MIN_TOKEN)
        if (!m_pifsIn->eof())
            m_pifsIn->getline(m_caBuffer+strlen(m_caBuffer), MAX_TOKEN, 0);

    if (strlen(m_caBuffer) > 0) {
        // check for spaces
        while (strchr(" \n\r\t", m_caBuffer[0]) > 0) {
            strcpy(caBuffer, m_caBuffer+1);
            strcpy(m_caBuffer, caBuffer);
            if (strlen(m_caBuffer) == 0)
                if (m_pifsIn->eof())

```

```

        break;
    else
        m_pifsIn->getline(m_caBuffer, MAX_TOKEN, 0);
}

// check for delimiters
if (strchr("+-*/()[ ]{}=&%!:.><;,#\"'\",m_caBuffer[0]) != NULL) {
    // one delimiter + one delimiter = one delimiter
    strcpy(caBuffer, m_caBuffer);
    caBuffer[2] = 0;
    if ((strstr(caBuffer, "++") != NULL) ||
        (strstr(caBuffer, "--") != NULL) ||
        (strstr(caBuffer, "+=") != NULL) ||
        (strstr(caBuffer, "-=") != NULL) ||
        (strstr(caBuffer, "*=") != NULL) ||
        (strstr(caBuffer, "/=") != NULL) ||
        (strstr(caBuffer, "%=") != NULL) ||
        (strstr(caBuffer, "!=") != NULL) ||
        (strstr(caBuffer, "==" ) != NULL) ||
        (strstr(caBuffer, "->") != NULL)) {
        strncpy(caReturn, m_caBuffer, 2);
        strcpy(caBuffer, m_caBuffer+2);
        strcpy(m_caBuffer, caBuffer);
        if (strlen(m_caBuffer) == 0)
            if (!m_pifsIn->eof())
                m_pifsIn->getline(m_caBuffer, MAX_TOKEN, 0);
    } else if (m_caBuffer[0] == '\\') {
        memset(caBuffer, 0, MAX_TOKEN);
        strncpy(caBuffer, m_caBuffer, 1);
        strcat(caReturn, caBuffer);
        strcpy(caBuffer, m_caBuffer+1);
        strcpy(m_caBuffer, caBuffer);
        do {
            if (strlen(m_caBuffer) == 0) {
                if (m_pifsIn->eof())
                    break;
                else
                    m_pifsIn->getline(m_caBuffer, MAX_TOKEN, 0);
            }
            if (m_caBuffer[0] == '\\\\') {
                strcpy(caBuffer, m_caBuffer+1);
                strcpy(m_caBuffer, caBuffer);
                memset(caBuffer, 0, MAX_BUFFER);
                strncpy(caBuffer, m_caBuffer, 1);
                strcat(caReturn, caBuffer);
                strcpy(caBuffer, m_caBuffer+1);
                strcpy(m_caBuffer, caBuffer);
            } else {
                memset(caBuffer, 0, MAX_BUFFER);
                strncpy(caBuffer, m_caBuffer, 1);
                strcat(caReturn, caBuffer);
                strcpy(caBuffer, m_caBuffer+1);
                strcpy(m_caBuffer, caBuffer);
            }
        } while (m_caBuffer[0] != '');
        memset(caBuffer, 0, MAX_BUFFER);
        strncpy(caBuffer, m_caBuffer, 1);
        strcat(caReturn, caBuffer);
        strcpy(caBuffer, m_caBuffer+1);
        strcpy(m_caBuffer, caBuffer);
    } else if (m_caBuffer[0] == '\\') {
    } else {
        strncpy(caReturn, m_caBuffer, 1);
        strcpy(caBuffer, m_caBuffer+1);
        strcpy(m_caBuffer, caBuffer);
        if (strlen(m_caBuffer) == 0)
            if (!m_pifsIn->eof())
                m_pifsIn->getline(m_caBuffer, MAX_TOKEN, 0);
    }
} else {
    while
    (strchr("abcdefghijklmnopqrstuvwxyzaBcDEFGHIJKLmNOPQRStUVWXYz1234567890_",m_caBuffer[0]) !=
    NULL) {
        memset(caBuffer, 0, MAX_BUFFER);
        strncpy(caBuffer, m_caBuffer, 1);
        strcat(caReturn, caBuffer);
        strcpy(caBuffer, m_caBuffer+1);
        strcpy(m_caBuffer, caBuffer);
    }
}

```

```

        if (strlen(m_caBuffer) == 0) {
            if (m_pifsIn->eof())
                break;
            else
                m_pifsIn->getline(m_caBuffer, MAX_TOKEN, 0);
        }
    }
}

if (strlen(caReturn) > 0) {
    pcReturn = new char [strlen(caReturn)+1];
    strcpy(pcReturn, caReturn);
} else
    pcReturn = NULL;

return pcReturn;
}

void CToken::Write(const char * pcOut) {
    *m_pofsOut << pcOut;
}

void CToken::Write(long lOut) {
    *m_pofsOut << lOut;
}

void CToken::Write(char cOut) {
    *m_pofsOut << cOut;
}

CToken& operator<<(CToken& os, const char * pcString) {
    os.Write(pcString);
    return os;
}

CToken& operator<<(CToken& os, long lLong) {
    os.Write(lLong);
    return os;
}

CToken& operator<<(CToken& os, char cChar) {
    os.Write(cChar);
    return os;
}

void CToken::PutBack(char * pcString) {
    char * pcTemp = new char [strlen(pcString) + strlen(m_caBuffer) + 2];

    Log("<\">%s\\"", pcString);

    strcpy(pcTemp, pcString);
    strcat(pcTemp, " ");
    strcat(pcTemp, m_caBuffer);

    strcpy(m_caBuffer, pcTemp);

    delete [] pcTemp;
}

CToken& operator>> ( CToken& is, char * & pcString) {
    char * pcOut = is.NextToken();
    if (pcOut == NULL)
        pcString[0] = 0;
    else {
        strcpy(pcString, pcOut);
        delete [] pcOut;
        is.Log("\">%s\\"", pcString);
    }
    return is;
}

CToken& operator>> ( CToken& is, char * pcString) {
    char * pcOut = is.NextToken();
    if (pcOut == NULL)
        pcString[0] = 0;
    else {
        strcpy(pcString, pcOut);
    }
}

```

```

        delete [] pcOut;
        is.Log("\'%s\'", pcString);
    }
    return is;
}

```

4.6. CToken.h

```

#ifndef INCLUDE_TOKEN
#define INCLUDE_TOKEN

#include <fstream.h>
#include <stdlib.h>

#define TOKEN_EOF NULL
#define MAX_TOKEN 200
#define MIN_TOKEN 100
#define MAX_BUFFER 400

class CToken {
public:
    CToken();
    CToken(char * pcInput, char * pcOutput, char * pcLog);
    ~CToken();
    void OpenOutput(char *);
    void OpenInput(char *);
    void OpenLog(char *);

    void Close();

    void PutBack(char *);
    char * NextToken();

    void Log(char *);
    void Log(char *, char *);
    void Log(char *, long);

    void Write(const char *);
    void Write(long);
    void Write(char);
private:
    ifstream * m_pifsIn;
    ofstream * m_pofsOut;
    ofstream * m_pofsLog;
    char m_caBuffer[MAX_BUFFER];
};

CToken & operator<<(CToken & os, const char * pcString);
CToken & operator<<(CToken & os, char cChar);
CToken & operator<<(CToken & os, long lLong);
CToken & operator>>(CToken & is, char * & pcString);
CToken & operator>>(CToken & is, char * pcString);
#endif

```

Filename: Diplom.doc
Directory: C:\WINNT\Profiles\d028838\Desktop\Diplom
Template: C:\users\templates\Harry\Harry.dot
Title: Diplomarbeit im Fachbereich Informatik an der Fachhochschule
für Technik und Gestaltung in Mannheim
Subject: Untersuchungen zum universellen Register-Maschinen-
Programm
Author: Harry Dietz
Keywords: Registermaschine, Register-Maschine, Turingmaschine,
universelles Programm
Comments:
Creation Date: 26.11.2000 14:20
Change Number: 46
Last Saved On: 01.12.2000 14:24
Last Saved By: SAP AG
Total Editing Time: 368 Minutes
Last Printed On: 02.12.2000 10:19
As of Last Complete Printing
Number of Pages: 113
Number of Words: 29.417 (approx.)
Number of Characters: 170.622 (approx.)