Eine Toolbox zum automatischen Erstellen von Übungsaufgaben für Rechnerarchitektur

Bachelorarbeit

von

Markus Kaserer

aus

Duisburg

vorgelegt am

Lehrstuhl für Rechnernetze
Prof. Dr. Martin Mauve
Heinrich-Heine-Universität Düsseldorf

November 2020

Betreuer:
Markus Brenneis, M. Sc.

Zusammenfassung

Die Aufgabe meiner Bachelorarbeit war die Entwicklung einer Anwendung zum automatischen Erstellen von Aufgaben für die Rechnerarchitektur. Die Anwendung sollte dazu dienen, den Mitarbeitern der Universität wertvolle Zeit beim Entwerfen von Übungsblättern zu ersparen.

Das Design der Anwendung war von Anfang an ein zentraler Aspekt. Die Architektur sollte so beschaffen sein, dass mögliche Erweiterungen in Zukunft einfach vorgenommen werden können. Deshalb gibt es in der Anwendung drei Komponenten. Die *Berechnung der Aufgaben*, die *Lösungserstellung* und die *Ausgabeformatierung*. Diese drei Komponenten funktionieren unabhängig voneinander. Wichtig ist dabei die Kommunikation über Schnittstellen, an deren Vorgaben sich die Komponenten halten müssen.

Ein weiterer wichtiger Aspekt war der Aufbau der Lösungen. Die Lösungen folgen einer Baumstruktur, damit die Ausgabeformatierung, unabhängig vom Inhalt der Lösung, immer dem gleichen Schema folgen kann. So ist es möglich neue Ausgabeformate zu implementieren, ohne genauer auf den Inhalt zu achten. Die Ausgabe erfolgt bisher im LaTeX-Format, da es das gängigste Format für das Erstellen von Übungsblättern im Fachbereich Informatik an der Heinrich-Heine-Universität ist.

Bereits implementiert sind fünf verschiedene Aufgabentypen. Dabei habe ich mich für das Umwandeln von Zahlensystemen, das Erstellen und Analysieren von Hammingcodes, die Berechnung von Hammingdistanzen, die Darstellung einer Dezimalzahl im IEEE-Standard 754 für Gleitkommazahlen und das Rechnen mit Binärzahlen im Zweierkomplement entschieden. Die Qualität der Algorithmen ist durch Unittests abgedeckt.

Die Anwendung ist für einen Mitarbeiter der Universität im Fachbereich Informatik leicht zu verstehen und auszuführen. Zur Unterstützung habe ich in einer *README*-Datei die nötigen Schritte und mögliche Funktionen aufgelistet. Damit ist die Anwendung bereits bei Abschluss meiner Arbeit ein hilfreiches Tool zum Erstellen von Aufgabenblättern der Rechnerarchitektur.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Kapitel 1

Einleitung

In meiner Abschlussarbeit habe ich mich mit Themen der Rechnerarchitektur befasst. Die Rechnerarchitektur ist in der Informatik das Teilgebiet, welches sich mit Aufbau und Organisation eines Computers beschäftigt. Der folgende Abschnitt gibt einen Einstieg in das Thema und vermittelt dem Leser Motivation und Ziele der Arbeit.

Im Kontext des Grundlagenmoduls Rechnerarchitektur an der Heinrich-Heine-Universität werden in jedem Semester, in dem das Modul unterrichtet wird, Aufgaben an die Studierenden gestellt. Diese Aufgaben dienen zum einen den Studierenden als Vorbereitung, zum anderen dienen sie als Zulassungsvoraussetzung für die Hauptklausur am Ende des Semesters. Das manuelle Anfertigen dieser Aufgaben inklusive Lösungsweg kostet die Mitarbeiter der Universität jedes Jahr viel Zeit. Die Motivation ist deshalb eine Lösung zu entwerfen, welche dieses Problem löst. Das Ziel meiner Arbeit ist daher die Entwicklung einer Anwendung, die die Anfertigung der Aufgaben und Lösungen automatisch durchführt. Als Anwender kommen dabei Mitarbeiter der Universität in Frage, welche mindestens einen Bachelor in Informatik absolviert haben. Daran orientieren sich die Details, wie z. B. das Eingabe- und Ausgabeformat. Ein wichtiger Aspekt beim Design der Anwendung wird die einfache Erweiterbarkeit sein. Das Hinzufügen neuer Aufgabentypen oder Ausgabeformate soll ohne großen Aufwand möglich sein, damit die Anwendung auch nach Abschluss meiner Arbeit angepasst und erweitert werden kann.

Die Arbeit ist neben der Einleitung in vier weitere Kapitel gegliedert. Im zweiten Kapitel werden theoretische Grundlagen vermittelt, auf denen die Algorithmen der Aufgaben basieren und die zum Verständnis der Arbeit beitragen. Im dritten Kapitel wird das Design und

die Architektur der Anwendung beschrieben und einzelne Entscheidungen und Probleme, die bei der Entwicklung enstanden sind, genauer betrachtet. Das vierte Kapitel dient als kurze Anleitung zur Ausführung der Anwendung. Es werden verschiedene Möglichkeiten zur Nutzung aufgezeigt. Das fünfte Kapitel bildet den Schluss. Es beinhaltet einen kurzen Rückblick und mögliche nächste Schritte.

Kapitel 2

Grundlagen

In diesem Kapitel wird das theoretische Wissen bereitgestellt, auf dem Teile der Arbeit basieren. Es werden die theoretischen Grundlagen für jene Aufgaben vermittelt, welche schließlich durch die Entwicklung einer Anwendung gelöst werden sollen. Dabei handelt es sich um die Themen *Gleitkommadarstellung*, *Umwandeln von Zahlensystemen*, *Fehlerkorrekturcodes* und *Rechnen mit Binärzahlen*.

2.1 Gleitkommadarstellung im IEEE-Standard 754

Es gibt unendlich viele reelle Zahlen. Um diese darzustellen, bräuchte der Computer unendlich viel Speicherplatz. Man hat sich daher für die Repräsentation von reellen Zahlen auf eine Beschränkung der Genauigkeit festgelegt. Das heißt, dass die gespeicherten Ziffern einer Zahl begrenzt werden müssen.

Es gibt verschiedene Gleitkommadarstellungen für Gleitkommazahlen. Im Kontext meiner Arbeit verwende ich den IEEE-Standard 754. Die Inhalte des folgenden Abschnitts basieren auf den Informationen aus [TA14, S. 699–705].

Man kann eine Gleitkommazahl n im Allgemeinen in der Notation

$$n = s \cdot m \cdot b^e$$

darstellen. Dabei ist b die Basis der Zahl, e der Exponent, s das Vorzeichen und m die Mantisse bzw. der rationale Anteil.

Beispielsweise gilt für die Dezimalzahl

$$4,15 = 0,415 \cdot 10^1$$

oder für die Binärzahl

$$11,1 = 0,111 \cdot 2^2.$$

Für die Darstellung von binären Gleitkommazahlen wurde der IEEE-Standard 754 eingeführt. Es gibt drei Formate. Im Folgenden beschäftigen wir uns mit der einfachen Genauigkeit, der Single Precision. Bei dieser Variante werden Gleitkommazahlen durch eine Folge von 32 Bit repräsentiert. Diese 32 Bit sind von links nach rechts wie folgt aufgeteilt:

- 1 Bit für das Vorzeichen
- 8 Bit für den Exponenten und
- 23 Bit für die Mantisse

Das Exponentensystem bei Single Precision speichert den Wert um den Bias $= 127$ erhöht. Man nennt das die Charakterisitk des Exponenten. Für den Exponenten 4 wird in den 8 Bits also der Wert $4 + 127 = 131$ bzw. 10000011 gespeichert. Der Grund dafür ist, dass der Exponent sowohl negativ als auch positiv sein kann. Statt negativer Werte speichert man Zahlen von 0 bis 255. Ein Exponent von -127 wird also als 0 gespeichert und ein Exponent von 128 als 255.

Die Mantisse wird normalisiert angegeben. Das bedeutet, dass sie immer mit 1,... beginnt. Der Vorteil davon ist, dass man die führende 1 nicht speichern muss und die Darstellung einheitlich ist.

2.1.1 Umwandlung einer Dezimalzahl in die binäre Gleitkommadarstellung mit Single Precision

Bei der Umwandlung einer Dezimalzahl in die binäre Gleitkommadarstellung mit Single Precision geht man wie folgt vor:

1. Das Vorzeichen ist entweder 0 (positiv) oder 1 (negativ).

2. Die Zahl vor dem Komma wird mit Hilfe der Divisionsrestmethode in eine Binärzahl umgewandelt.

3. Die Zahl hinter dem Komma wird mit Hilfe der Multiplikationsmethode in eine Binärzahl umgewandeln.

4. Die beiden Ergebnisse werden zusammengeführt und normalisiert. Man erhält die Form 1,... . Die Zahlenfolge hinter dem Komma wird als Mantisse gespeichert.

5. Der Exponent ergibt sich aus der Anzahl an Stellen, um die beim Normalisieren verschoben wurde, plus 127. Wird das Komma nach links verschoben, ist der Exponent positiv. Beim Verschieben nach rechts wird er negativ. Der Exponent wird ebenfalls mit Hilfe der Divisionsrestmethode als Binärzahl gespeichert.

Die Zahl im IEEE-Standard 754 ergibt sich nun aus Vorzeichen, Exponent und Mantisse.

Beispiel

$-2,4_{10}$ in die binäre Gleitkommadarstellung mit Single Precision umwandeln.

1. Das Vorzeichen ist negativ, also 1.

2. Die Zahl vor dem Komma ist

$$2_{10} = 10_2$$

3. Für die Zahl nach dem Komma verwendet man die Multiplikationsmethode:

$$
\begin{aligned}
0.4 \cdot 2 &= 0.8 \quad 0 \\
0.8 \cdot 2 &= 1.6 \quad 1 \\
0.6 \cdot 2 &= 1.2 \quad 1 \\
0.2 \cdot 2 &= 0.4 \quad 0 \\
0.4 \cdot 2 &= 0.8 \quad 0 \\
0.8 \cdot 2 &= 1.6 \quad 1 \\
0.6 \cdot 2 &= 1.2 \quad 1 \\
0.2 \cdot 2 &= 0.4 \quad 0 \\
0.4 \cdot 2 &= 0.8 \quad 0 \\
0.8 \cdot 2 &= 1.6 \quad 1 \\
&\vdots \\
\Rightarrow 0,&0110011001100110011001_2 \ldots
\end{aligned}
$$

4. Beide Zahlen zusammen ergeben

$$2,4_{10} = 10,0110011001100110011001_2 \ldots$$

Normalisiert erhält man

$$10,0110011001100110011001_2 \ldots \cdot 2^0 = 1,0011001100110011001100_2 \ldots \cdot 2^1.$$

Die Zahlenfolge hinter dem Komma wird als Mantisse gespeichert.

5. Da bei der Normalisierung um 1 Stelle nach links verschoben wurde, erhält man für die Charakteristik des Exponenten

$$\text{Charakteristik} = \text{Exponent + Bias} = 1 + 127 = 128 \text{ und } 128_{10} = 10000000_2$$

Schließlich ist das Ergebnis

Vorzeichen	Exponent	Mantisse
1	10000000	00110011001100110011001

2.1.2 Rundungsfehler

Da die Genauigkeit durch die Begrenzung auf 23 Bit in der Mantisse eingeschränkt ist, entstehen Rundungsfehler. Ist eine Zahl nicht exakt darstellbar, rundet man auf die nächsthöhere oder -kleinere darstellbare Zahl. Im Standard nimmt man dann die näherliegende Zahl, ähnlich dem Runden bei Dezimalzahlen.

2.2 Fehlerkorrekturcodes

In Computern kommt es manchmal zu Fehlern. Dabei können Daten verändert oder im schlimmsten Fall verloren gehen. Um solche Fehler zu erkennen und gegebenenfalls zu korrigieren, gibt es Fehlerkorrekturcodes. Codewörter bestehen aus gespeicherten Datenbits und hinzugefügten Prüfbits. Durch die hinzugefügten Prüfbits kann der Computer beim Lesen von Daten mögliche Fehler erkennen. Um ein Datenwort zu sichern, kann man den Hammingcode benutzen. Mit Hilfe der Hammingdistanz kann man den Hammingcode sowie andere Codes charakterisieren.

2.2.1 Hammingdistanz

Der Abschnitt zur Hammingdistanz basiert auf den Informationen aus [TA14, S. 97]. Die Anzahl der Bitstellen, an denen sich zwei Codewörter unterscheiden, gibt die Hammingdistanz an. Für die beiden Codewörter $c_1 = 10001001$ und $c_2 = 10110001$ beträgt sie beispielsweise 3, was man aus folgender Tabelle ablesen kann:

1	0	0	0	1	0	0	1
1	0	①	①	⓪	0	0	1

Betrachtet man mehr als zwei Codewörter, wird die Hammingdistanz aller Codewörter paarweise berechnet und das Minimum davon bestimmt. Dies nennt man dann Hammingdistanz eines Codes.

Die Hammingdistanz der drei Codewörter $c_1 = 0000$, $c_2 = 0110$ und $c_3 = 1101$ beträgt 2, da sich c_1 und c_2 an 2, c_1 und c_3 an 3 und c_2 und c_3 ebenfalls an 3 Bitstellen unterscheiden.

0	0	0	0
0	①	①	0

Die Hammingdistanz zwischen 0000 und 0110 beträgt 2.

0	0	0	0
①	①	0	①

Die Hammingdistanz zwischen 0000 und 1101 beträgt 3.

0	1	1	0
①	1	⓪	①

Die Hammingdistanz zwischen 0110 und 1101 beträgt 3.

2.2.2 Hammingcode

Dieser Abschnitt zum Hammingcode basiert auf den Informationen aus [TA14, S. 97–100]. Der Hammingcode dient dazu in einem Codewort alle Ein-Bit-Fehler zu erkennen. Dafür werden einem Datenwort mit m Datenbits r Prüfbits hinzugefügt, sodass sich ein Codewort mit $m+r$ Bits ergibt. Beim Hammingcode haben alle möglichen Codewörter eine Hammingdistanz von 3. Daher ist es möglich alle Zwei-Bit-Fehler zu erkennen und alle Ein-Bit-Fehler zu korrigieren.

Die Nummerierung der Bits im Codewort beginnt bei 1 und endet bei $m+r$. Die Prüfbits verteilen sich auf die Positionen, die eine Zweierpotenz sind, also 1, 2, 4, 8, 16, … . Auf die übrigen Plätze werden die Datenbits verteilt.

Jedes Prüfbit p_i wird nun gesetzt. Dabei addiert man alle Bits im Codewort, deren Bitposition in der binären Darstellung eine 1 an der i-ten Stelle von rechts haben, und rechnet das Ergebnis modulo 2. Je nachdem, ob mit gerader oder ungerader Parität gesichert wird, wird das Prüfbit gesetzt.

- Für p_1 werden die Bits an den Positionen 1, 3, 5, 7, 9, 11, 13, 15, … betrachtet.
- Für p_2 werden die Bits an den Positionen 2, 3, 6, 7, 10, 11, 14, 15, … betrachtet.

- Für p_3 werden die Bits an den Positionen 4, 5, 6, 7, 9, 12, 13, 14, 15, ... betrachtet.
- usw.

Bei der Fehlererkennung eines Codewortes wird geprüft, ob alle Prüfbits den korrekten Wert haben. Ist dies der Fall, liegt kein bzw. mehr als ein Fehler vor. Andernfalls addiert man die Positionen der Prüfbits, welche einen falschen Wert haben. Das Ergebnis gibt die Position des Fehlers an. Sind zum Beispiel p_1 (Position 1) und p_3 (Position 4) falsch gesetzt, liegt der Fehler an Position $1+4=5$.

Die Aufteilung eines Codewortes der Länge 13 mit 9 Datenbits und 4 Prüfbits ist in Tabelle 2.1 dargestellt.

Codewortbits	c_1	c_2	c_3	c_4	c_5	c_6	c_7	c_8	c_9	c_{10}	c_{11}	c_{12}	c_{13}
Daten- und Prüfbits	p_1	p_2	d_1	p_3	d_2	d_3	d_4	p_4	d_5	d_6	d_7	d_8	d_9
Bereich des Prüfbits p_1	X		X		X		X		X		X		X
Bereich des Prüfbits p_2		X	X			X	X			X	X		
Bereich des Prüfbits p_3				X	X	X	X					X	X
Bereich des Prüfbits p_4								X	X	X	X	X	X

Tabelle 2.1: Aufteilung eines Hammingcodes der Länge 13 mit 9 Datenbits und 4 Prüfbits

Je nach Länge des Codewortes, braucht es eine Mindestanzahl an Prüfbits, um alle möglichen Ein-Bit-Fehler zu erkennen. Nach [TA14, S. 98] ist die Bedingung für die minimale Anzahl an Prüfbits durch die Ungleichung

$$m + r + 1 \leq 2^r$$

gegeben. In Tabelle 2.2 aus [TA14, S. 98] findet man diese Anzahl für verschiedene Wortlängen.

Anzahl Datenbits	Anzahl Prüfbits
8	4
16	5
32	6
64	7
128	7
256	8
512	9

Tabelle 2.2: Verschiedene Werte für die Anzahl an Daten- und Prüfbits

2.3 Umwandlung von Zahlensystemen

Im Computer werden Zahlen in anderen Zahlensystemen als dem geläufigen Dezimalsystem gespeichert. Deshalb ist es wichtig diese Systeme zu kennen und mit ihnen umgehen zu können. Die wichtigsten Systeme sind das Binär-, das Oktal- und das Hexadezimalsystem. In der Rechnerarchitektur ist die Umwandlung zwischen diesen Systemen relevant.

Für die Arbeit sind zwei Umwandlungsarten wichtig. Die erste Art ist die Umwandlung von einer Dezimalzahl in eine Binär- oder Hexadezimalzahl. Die zweite Art ist die Umwandlung einer Binär- oder Hexadezimalzahl in eine Dezimalzahl. Da das Vorgehen für Binär- und Hexadezimalzahlen identisch ist, gelten die folgenden Algorithmen für beide Fälle. Die Algorithmen basieren auf den Grundlagen aus [TA14, S. 689–691].

2.3.1 Umwandlung einer Dezimalzahl in eine Binär- oder Hexadezimalzahl

Um eine beliebige Dezimalzahl in eine Binär- oder Hexadezimalzahl umzuwandeln, nutzt man folgenden Algorithmus:

1. Die Zahl wird durch den Wert der Basis, welche die neue Zahl haben wird, geteilt. In diesem Fall ist die Basis 2 (binär) oder 16 (hexadezimal).

2. Der dabei entstandene Rest wird gespeichert und die Zahl mit dem Wert, der bei der

Division entsteht, aktualisiert.

3. Die Schritte eins und zwei werden so lange wiederholt, bis der Quotient Null ergibt.

4. Die entstandenen Restwerte von unten nach oben gelesen ergeben die Zahl zur neuen Basis.

Beispiele

1. 26_{10} in eine Binärzahl umwandeln:

$$\begin{aligned} 26 \div 2 &= 13 \quad 26 \bmod 2 = 0 \\ 13 \div 2 &= 6 \quad 13 \bmod 2 = 1 \\ 6 \div 2 &= 3 \quad 6 \bmod 2 = 0 \\ 3 \div 2 &= 1 \quad 3 \bmod 2 = 1 \\ 1 \div 2 &= 0 \quad 1 \bmod 2 = 1 \end{aligned}$$

$\Longrightarrow 26_{10} = 11010_2$

2. 26_{10} in eine Hexadezimalzahl umwandeln:

$$\begin{aligned} 26 \div 16 &= 1 \quad 26 \bmod 16 = \mathrm{A}_{16} \\ 1 \div 16 &= 0 \quad 1 \bmod 16 = 1_{16} \end{aligned}$$

$\Longrightarrow 26_{10} = 1\mathrm{A}_{16}$

2.3.2 Umwandlung einer Binär- oder Hexadezimalzahl in eine Dezimalzahl

Um eine beliebige Binär- oder Hexadezimalzahl a der Form $a = a_0 a_1 \ldots a_n$ zur Basis b in eine Dezimalzahl umzuwandeln, nutzt man die Formel

$$\sum_{i=0}^{n} a_i \cdot b^{n-i} = a_0 \cdot b^n + a_1 \cdot b^{n-1} + \cdots + a_n \cdot b^0.$$

Beispiele

1. $1A_{16} = 1 \cdot 16^1 + 10 \cdot 16^0 = 16 + 10 = 26_{10}$
2. $11010_2 = 1 \cdot 2^4 + 1 \cdot 2^3 + 0 \cdot 2^2 + 1 \cdot 2^1 + 0 \cdot 2^0 = 16 + 8 + 2 = 26_{10}$

2.4 Rechnen im Zweierkomplement

Der folgende Abschnitt basiert auf den Informationen aus [TA14, S. 691–693]. Für die Darstellung von negativen Binärzahlen wird häufig das Zweierkomplement benutzt. Hier werden sie auf 8 Bits beschränkt. Das erste Bit gibt dabei das Vorzeichen an. 0 für positive Zahlen und 1 für negative Zahlen. Übrig bleiben 7 Bits für den Wert der Zahl. Daher ist der Wertebereich für 8-Bit Zahlen im Zweierkomplement beschränkt durch -128 nach unten und 127 nach oben.

Die Regeln für die Addition von binären Zahlen lauten:

- $0 + 0 = 0$, kein Übertrag.
- $0 + 1 = 1$, kein Übertrag.
- $1 + 0 = 1$, kein Übertrag.
- $1 + 1 = 0$, 1 als Übertrag.

Bei der Subtraktion wird die entsprechende Zahl in ihre negative Darstellung im Zweierkomplement umgewandelt und addiert. Die Regeln ändern sich nicht.

Negieren einer Binärzahl im Zweierkomplement

Das Negieren einer Binärzahl im Zweierkomplement folgt dabei zwei Schritten, unabhängig davon, ob man eine negative Zahl in eine positive umwandeln möchte oder anders herum.

1. Jede 1 wird zu einer 0 umgedreht und jede 0 zu einer 1.

2. Zum Ergebnis wird 1 hinzuaddiert.

Überlauf

Ein Überlauf entsteht bei der Addition im Zweierkomplement, wenn das Ergebnis nicht mehr im gültigen Wertebereich liegt. Das bedeutet, die Zahl ist nicht mehr darstellbar. Bei der Addition im Zweierkomplement kommt es nur zum Überlauf, wenn sich der Übertrag in das Vorzeichenbit vom Übertrag aus dem Vorzeichenbit heraus unterscheidet. Ansonsten spielt der Übertrag aus dem Vorzeichenbit heraus für die Berechnung keine Rolle.

Man kann dies an folgenden Beispielen gut erkennen:

$$(-127_{10}) + (-1_{10}) = (-128_{10})$$

$$\begin{array}{cccccccccc}
 & 1 & 0 & 0 & 0 & & 0 & 0 & 0 & 1 \\
+ & 1 & 1 & 1 & 1 & & 1 & 1 & 1 & 1 \\
\hline
1 & 1 & 1 & 1 & 1 & & 1 & 1 & 1 & \\
\hline
 & 1 & 0 & 0 & 0 & & 0 & 0 & 0 & 0
\end{array}$$

Es findet ein Übertrag sowohl in das Vorzeichenbit als auch aus dem Vorzeichenbit heraus statt. Der Übertrag wird ignoriert und das Ergebnis ist korrekt.

$$(-127_{10}) + (-2_{10}) = (-129_{10})$$

$$\begin{array}{cccccccccc}
 & 1 & 0 & 0 & 0 & & 0 & 0 & 0 & 1 \\
+ & 1 & 1 & 1 & 1 & & 1 & 1 & 1 & 0 \\
\hline
1 & & & & & & & & & \\
\hline
 & 0 & 1 & 1 & 1 & & 1 & 1 & 1 & 1
\end{array}$$

Es findet nur ein Übertrag aus dem Vorzeichenbit heraus statt. Das Ergebnis ist falsch, da -129 nicht dargestellt werden kann.

Beispiel Subtraktion im Zweierkomplement

Folgendes Beispiel basiert auf [TA14, S. 693].

$10_{10} + (-3)_{10} = 7_{10}$

Zunächst negiert man $3_{10} = 0000\ 0011_2$. Alle Zahlen invertieren ergibt $1111\ 1100_2$. Eins addiert erhält man $1111\ 1101_{2c}$. Es folgt die Rechnung:

	0	0	0	0	1	0	1	0	(10_{10})
+	1	1	1	1	1	1	0	1	(-3_{10})
1	1	1	1	1	0	0	0		Übertrag
	0	0	0	0	0	1	1	1	(7_{10})

Kapitel 3

Design und Architektur der Anwendung

In diesem Kapitel wird das Design und die Architektur der Anwendung vorgestellt. Dabei gehe ich zunächst auf den grundlegenden Aufbau der Anwendung ein und erläutere daraufhin, welche Teilentscheidungen und Probleme aufgetreten sind.

3.1 Grundlegendes Design

Die Anwendung ist, wie in Abbildung 3.1 dargestellt, in die drei Komponenten *Berechnung*, *Ausgabeformatierung* und *Lösungserstellung* aufgeteilt.

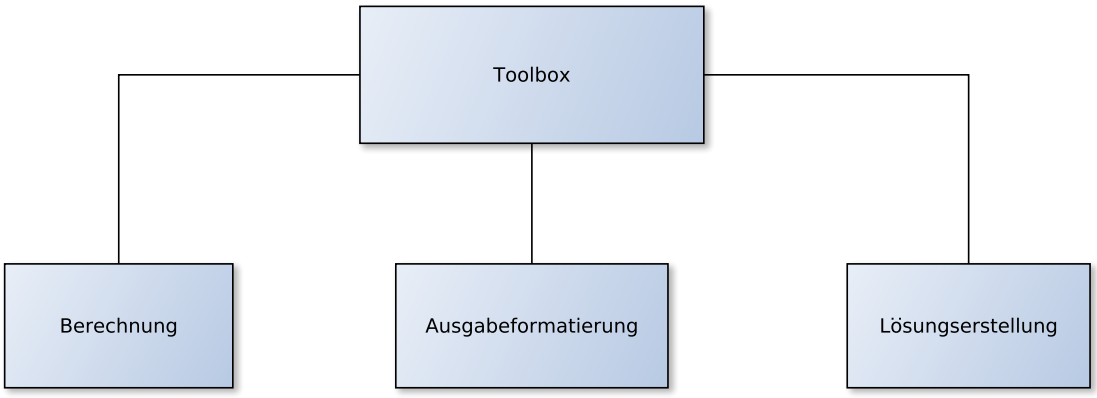

Abbildung 3.1: Gundlegendes Design der Anwendung

Die Komponente *Berechnung* kümmert sich um die algorithmischen Berechnungen der ein-

zelnen Aufgabentypen. Es sind fünf Aufgabentypen bereits implementiert. Dabei handelt es sich um das Umwandeln von Zahlensystemen, Erstellen und Analysieren von Hammingcodes, Berechnung von Hammingdistanzen, Darstellung einer Dezimalzahl im IEEE-Standard 754 für Gleitkommazahlen und das Rechnen mit Binärzahlen im Zweierkomplement.

Danach kommt die Komponente *Lösungserstellung* ins Spiel. Aus den berechneten Werten wird eine Lösung erstellt. Dabei wird eine Baumstruktur verwendet, welche die Ausgabeformatierung vereinfachen soll. Jede Lösung beinhaltet sowohl Aufgabenstellung als auch Lösungsweg. Die Struktur der Lösungsbäume wird in Abschnitt 3.2.2 ausführlich betrachtet.

Anschließend sorgt die Komponente *Ausgabeformatierung* dafür, dass aus dem entstandenen Lösungsbaum das gewünschte Ausgabeformat entsteht. Bisher implementiert ist die Ausgabe im LaTeX-Format.

Die Klassen, die für die Realisierung der Komponenten *Berechnung*, *Lösungserstellung* und *Ausgabeformatierung* zuständig sind, befinden sich in den Packages *exercises*, *solutiontrees* und *outputgeneration*. Zusätzlich gibt es ein weiteres Package *nodes*. Dieses enthält Knoten, aus denen der Lösungsbaum besteht.

Folgende Knotentypen gibt es bisher:

- Text
- Base
- Number
- Operator
- Section
- Power
- HLine (horizontal line)
- MathFormula

- FloatingPointNumber

3.2 Entscheidungen und Probleme

In diesem Abschnitt werden Probleme und Teilentscheidungen erläutert, die bei der Entwicklung der Anwendung aufgetreten sind. Die wichtigsten Entscheidungen waren die Sicherstellung der einfachen Erweiterbarkeit, der Strukturaufbau der Lösungen, die Abwägung von Vor- und Nachteilen des Visitor Pattern, die Vermeidung von Rundungsfehlern bei Gleitkommazahlen und die Sicherstellung der Qualität der implementierten Algorithmen durch Unittests.

3.2.1 Erweiterbarkeit

Beim Design der Anwendung war der Aspekt der einfachen Erweiterbarkeit besonders wichtig. Die einzelnen Komponenten müssen möglichst unabhängig voneinander arbeiten, sodass Erweiterungen für jede Komponente unkompliziert zu realisieren sind. Beim Hinzufügen eines neuen Aufgabentypen darf das Ausgabeformat beispielsweise keine Rolle spielen.

Um diesen Aspekt sicherzustellen, gibt es für jede Komponente (Berechnungen, Lösungserstellung, Outputgenerator) ein Interface als Schnittstelle. Über diese Schnittstelle kommunizieren die Komponenten miteinander.

Möchte der Nutzer beispielsweise einen neuen Aufgabentypen hinzufügen, muss er sich in erster Linie darum kümmern die Berechnungen durchzuführen. Das Erstellen der Lösung ist durch die bereits implementierten Knotentypen einfach umzusetzen. Es gibt jedoch immer die Möglichkeit, dass neue Knotentypen nötig sind. Diese müssen dann ergänzt werden. Der Nutzer muss sich sonst nur an die vorgegebene Baumstruktur der Lösungsbäume halten, wie in 3.2.2 beschrieben.

3.2.2 Struktur der Lösung

Innerhalb der Anwendung wird die Lösung einer Aufgabe als Baumstruktur aufgebaut. Die Struktur ähnelt dabei der eines HTML-DOM-Baums. Der Vorteil dabei ist, dass die Ausgabeformatierung nur die Wurzel eines solchen Baumes bekommen muss. Die Verarbeitung des restlichen Baumes folgt dann bestimmten Regeln, die unabhängig vom Inhalt der Lösung gelten. Die Ausgabeformatierung durchläuft den Lösungsbaum in der Reihenfolge einer Tiefensuche von links nach rechts. Dabei wird jeder Knoten mit Hilfe einer Funktion in das gewünschte Format umgewandelt.

Jede Lösung besteht grob betrachtet aus den drei Teilen Aufgabenstellung (*exercise*), Details zur Aufgabenstellung (*subexercise*) und dem Lösungsteil (*loesung*). Der Baum einer Lösung kann beispielsweise so aussehen wie in Abbildung 3.2.

```
root
├─ Section(exercise)
│  └─ Text (Text der Aufgabenstellung)
├─ Section(subexercise)
│  └─ MathFormula
└─ Section(loesung)
   ├─ Text (Einführungssatz)
   ├─ Tabelle (Tabelle enthält MathFormula Objekte)
   │  └─ MathFormula
   ├─ Text (Zwischenergebnis)
   └─ Text (Schlussfolgerung)
```

Abbildung 3.2: Beispielstruktur eines Lösungsbaumes

3.2.3 Visitor Pattern vs Dependency Inversion

Für jeden Knoten des Lösungsbaums ist festgelegt, wie seine Repräsentation in einem bestimmten Ausgabeformat aussieht. Bisher ist das Ausgabeformat in LaTeX implementiert. Bei der Entscheidung, wie die Umwandlung in das gewünschte Ausgabeformat geschehen soll, gab es zwei Möglichkeiten.

Die erste Möglichkeit war Dependency Inversion. Jede spezielle Knotenklasse erweitert eine allgemeine Knotenklasse und muss dabei eine Funktion implementieren, die sich um die

Ausgabe kümmert.

Die zweite Möglichkeit war das Visitor Pattern. Bei diesem Designprinzip gibt es für jedes Ausgabeformat eine Besucherklasse, in der für jeden Knotentyp die Umwandlung in das Ausgabeformat implementiert ist.

In Abbildung 3.3 ist Variante eins dargestellt. Der LatexPrinter ruft in jeder Knotenklasse eine Funktion auf, die das LaTeX-Format erstellt und zurückgibt. Die Implementierung der Funktion ist in den Knotenklassen selbst umgesetzt. Der LatexPrinter kümmert sich nur um die Ausführung.

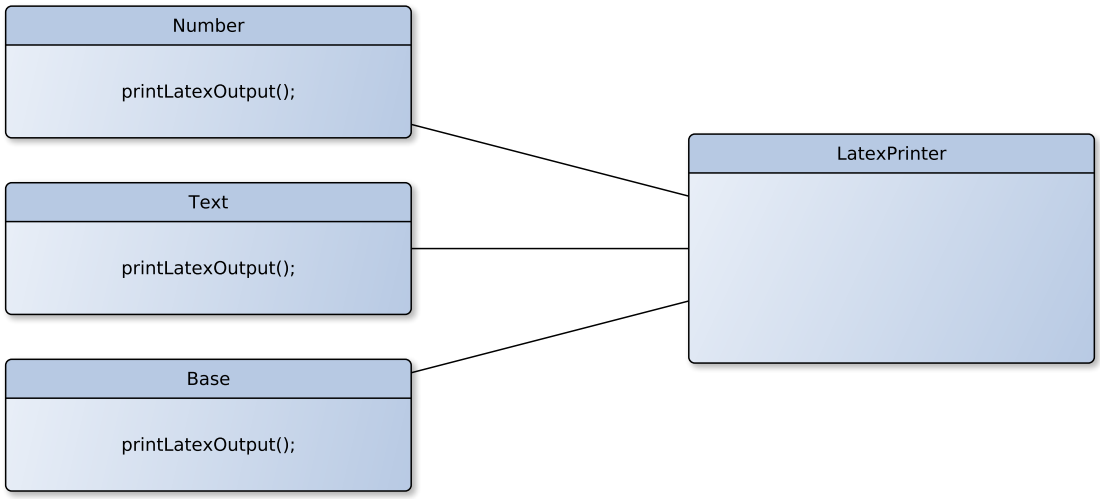

Abbildung 3.3: Beispiel LatexPrinter ruft Methode in jeder Klasse auf

In Abbildung 3.4 ist Variante zwei dargestellt. Der LatexPrinter ist nun ein Besucher. Er besucht also jeden Knoten einmal. Dafür ist in jeder Knotenklasse eine Funktion implementiert, die den Besucher (hier den LatexPrinter) entgegen nimmt. Der Knoten wiederum ruft eine Funktion in der Besucherklasse auf und übergibt sich selbst als Parameter. Für jeden Knotentypen ist in der Besucherklasse die Umwandlung in das gewünschte Ausgabeformat implementiert.

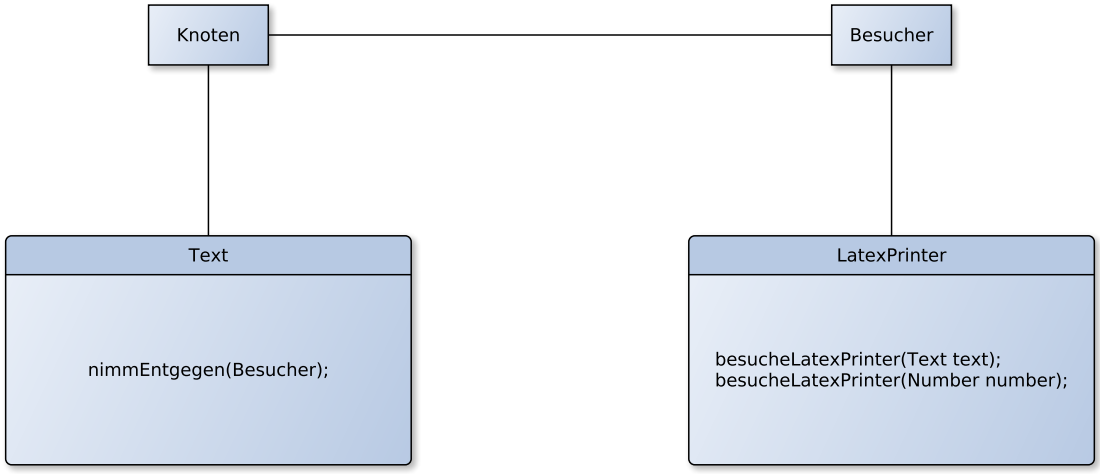

Abbildung 3.4: Beispiel Visitor Pattern

Der Vorteil bei der Verwendung des Visitorpattern ist das einfache Hinzufügen von neuen Besuchern. In unserem Fall entspricht das einem neuen Ausgabeformat. Dazu muss lediglich ein neuer Besucher angelegt werden, der für jeden Knotentypen die entsprechende Funktion implementiert. Das Hinzufügen von neuen Knotentypen ist bei dieser Variante jedoch deutlich aufwändiger, da viele zusätzliche Funktionen in unterschiedlichen Klassen implementiert werden müssen.

Bei Dependency Inversion wird zudem das Open-Closed-Prinzip eingehalten. Die Anwendung ist offen für Erweiterungen durch neue Klassen. Dadurch ändert sich aber nichts an dem grundlegenden Verhalten. Änderungen müssen außerdem nur in niedrigeren Ebenen vorgenommen werden, hier in den einzelnen Funktionen der Knotenklassen.

Da es im Kontext der Arbeit dazu kommen wird, dass neue Aufgabentypen und damit auch neue Knotentypen hinzugefügt werden, habe ich mich gegen das Visitor Pattern entschieden. Die Implementierung von mehreren neuen Ausgabeformaten ist in Zukunft nicht besonders wahrscheinlich und daher überwiegen die Nachteile bei dieser Variante.

3.2.4 Rundungsfehler bei Gleitkommazahlen

Wie in Kapitel 2.1 erläutert, kommt es bei der Darstellung von Gleitkommazahlen aufgrund der binären Repräsentation zu Rundungsfehlern. Da manche reelle Zahlen wie $\frac{1}{3}$ unendlich lang sind oder andere reelle Zahlen aufgrund der Beschränkung der Genauigkeit nicht exakt darstellbar sind, kommt es bei Berechnungen zu Rundungsfehlern.

Teilt man beispielsweise 1 durch 3 und multipliziert das Ergebnis anschließend wieder mit 3, müsste man wiederum 1 erhalten, da

$$1 \div 3 = \frac{1}{3} = 0,33333\ldots \text{ und } 0,33333\ldots \cdot 3 = \frac{1}{3} \cdot 3 = 1.$$

Der Computer allerdings kann keine unendlich lange Zahl speichern und beschränkt die Nachkommastellen. Er speichert das Zwischenergebnis z. B. als $0,33333$. Führt man also die gleiche Rechnung im Computer durch, erhält man

$$1 \div 3 = 0,33333 \text{ und } 0,33333 \cdot 3 = 0,99999 \neq 1.$$

Bei der Implementierung des Verfahrens zur Umwandlung einer Dezimalzahl in die binäre Gleitkommadarstellung kam es häufig zu diesen Fehlern. Um solche Rundungsfehler zu vermeiden, war eine Überlegung die Java-Klasse BigInteger zu benutzen und die Zahlen so zu vergrößern, dass ohne Kommas gerechnet werden kann. Die Klasse speichert die Zahl 7,4638748832 dann beispielsweise als 74638748832 und rechnet mit der ganzen Zahl weiter. Am Ende kann die Zahl wieder umgewandelt werden, sodass es keine Rundungsfehler gibt.

Da die Klasse BigDecimal allerdings genau diese Umrechnungen übernimmt und zudem viele Operationen mit mehreren BigDecimal-Objekten bereits implementiert sind, habe ich mich für diesen Datentypen entschieden.

3.2.5 Tests

Um die Qualität der implementierten Algorithmen sicherzustellen, gibt es für jeden Aufgabentypen Unittests. Die Tests prüfen die Algorithmen mit verschiedenen Eingaben und

decken alle möglichen Spezialfälle ab.

Eine Ausnahme bildet dabei die Aufgabe *Umwandlung von Zahlensystemen*, da für die Berechnungen innerhalb dieser Klasse lediglich Funktionen benutzt wurden, die von Java zur Verfügung gestellt sind und damit eine Korrektheit der Implementierung angenommen werden kann.

3.2.6 Periodenerkennung bei der Berechnung der Gleitkommadarstellung

Den Algorithmus, der die Umwandlung einer Dezimalzahl in die Gleitkommadarstellung beschreibt, findet man in Abschnitt 2.1.1. Bei Schritt 3, dem Umwandeln der Zahl hinter dem Komma in eine Binärzahl, kann es zu Perioden kommen. Wenn sich ein Zwischenergebnis der Multiplikationsmethode wiederholt, wiederholen sich auch die darauffolgenden Schritte. In dem Fall kann die weitere Berechnung abgebrochen werden.

In der Anwendung ist die Erkennung von Perioden nicht implementiert, da die Erparnis, anders als bei der händischen Berechnung, im Computer keine signifikante Größe hat.

Kapitel 4

Benutzung der Anwendung

In diesem Kapitel wird die Nutzung der Anwendung erklärt. Es gibt eine Übersicht darüber, welche Parameter für welchen Aufgabentypen übergeben werden müssen.

Der Ort der Ausgabedatei kann in der Klasse *Main* angepasst werden. Da die Zielgruppe der Anwendung Personen mit Programmierkenntnissen sind, ist diese Anpassung im Code ausreichend.

4.1 Ausführung unter Linux

Unter dem Link

`https://gitlab.cs.uni-duesseldorf.de/cn-tsn/students/bachelor/ba-kaserer-code`

kann der Sourcecode heruntergeladen oder geklont werden.

Die Anwendung ist als Gradle-Projekt angelegt. Man kann zunächst über die Konsole *gradle build* aufrufen, allerdings wird der Befehl bei jedem Ausführen automatisch durchgeführt.

Das Ausführen der Anwendung erfolgt mit dem Konsolenbefehl

gradle run -PmyArgs="<Liste mit Parametern>"

Der erste Parameter gibt den Aufgabentypen vor. Ein Überblick über die restlichen Parameter für jeden Aufgabentypen folgt. Am Ende des Kapitels findet man ein Ausgabebeispiel von verschiedenen Aufrufen.

4.1.1 Ausführung Umwandlung von Zahlensystemen

Der Aufruf erfolgt mit den Parametern in dieser Reihenfolge:

baseconversion <umzuwandelnde Zahl> <Ausgangsbasis> <Zielbasis>

Die Umwandlung funktioniert mit positiven Zahlen.

Beispiel

gradle run -PmyArgs="baseconversion,110010101,2,10"

4.1.2 Ausführung Hammingcode

Der Aufruf erfolgt mit den Parametern in dieser Reihenfolge:

hammingcode <Inputwort> <Anzahl Datenbits> <Anzahl Paritätbits> <Parität> <Input-typ>

Dabei ist das Inputwort ein Datenwort oder ein Codewort. Im Parameter Inputtyp wird dies angegeben mit dataword oder codeword. Die Parität ist entweder even oder odd.

Beispiel

gradle run -PmyArgs="hammingcode,01110010011,11,4,even,dataword".

4.1.3 Ausführung Hammingdistanz

Der Aufruf erfolgt mit den Parametern in dieser Reihenfolge:

hammingdistance <Codewort1> <Codewort2> ... <CodewortN>

Es wird die Hammingdistanz für eine beliebige Anzahl binärer Codewörter berechnet.

Beispiel

gradle run -PmyArgs="hammingdistance,0000,0110,1011"

4.1.4 Ausführung Gleitkommadarstellung

Der Aufruf erfolgt mit den Parametern in dieser Reihenfolge:

singleprecision <Eingabezahl>

Die Eingabezahl ist eine Dezimalzahl, wobei das Komma durch einen Punkt gekennzeichnet wird. Die Zahl wird in die binäre IEEE 754 Gleitkommadarstellung in Single Precision (32 Bit) umgewandelt.

Beispiel

gradle run -PmyArgs="singleprecision,-18.4"

4.1.5 Ausführung Rechnen im Zweierkomplement

Der Aufruf erfolgt mit den Parametern in dieser Reihenfolge:

twoscomplement <Zahl1> <Zahl2>

Die zahlen werden als Dezimalzahlen übergeben.

Beispiel

gradle run -PmyArgs="twoscomplement,55,-15"

Die LaTeX-Dateien jeder einzelnen Aufgabe müssen nach der Generierung in ein Dokument eingefügt werden, welches den Rahmen eines Übungsblattes bereitstellt. Eine Vorlage für ein solches Dokument wird im Rahmen der Anwendung bereitgestellt. Es beruht auf der Entwicklung von früheren Arbeitsblättern der Rechnerarchitektur.

Die oben genannten Beispielaufrufe in einer Datei zusammengefasst sieht man in den Abbildungen 4.1, 4.2 und 4.3. Ein Aufgabenblatt mit Lösungswegen könnte letztendlich so aussehen.

hhu. **Institut für Informatik**

SoSe 2019
Blatt 2
Abgabe Pool

Rechnerarchitektur
Beispiellösung zur Übung

1 Testausgaben

Aufgabe 1 *Umrechnen von Zahlensystemen* (Punkte)
Wandeln Sie die folgende Zahl zur Basis 2 in eine Zahl zur Basis 10 um.

(a) 110010101_2

Lösungsvorschlag: *(Punkte)*

$$110010101_2 = 1 \cdot 2^8 + 1 \cdot 2^7 + 0 \cdot 2^6 + 0 \cdot 2^5 + 1 \cdot 2^4 + 0 \cdot 2^3 + 1 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0 = 405_{10}$$

Aufgabe 2 *Hammingcode* (Punkte)
Sichern Sie das folgende Datenwort mit einem (11,4)-Hammingcode mit gerader Parität.

(a) 01110010011_2

Lösungsvorschlag: *(Punkte)*

Codewortbits	c_1	c_2	c_3	c_4	c_5	c_6	c_7	c_8	c_9	c_{10}	c_{11}	c_{12}	c_{13}	c_{14}	c_{15}
Daten- und Prüfbits	p_1	p_2	d_1	p_3	d_2	d_3	d_4	p_4	d_5	d_6	d_7	d_8	d_9	d_{10}	d_{11}
Bereich des Prüfbits p_1	X		X		X		X		X		X		X		X
Bereich des Prüfbits p_2		X	X			X	X			X	X			X	X
Bereich des Prüfbits p_3				X	X	X	X					X	X	X	X
Bereich des Prüfbits p_4								X	X	X	X	X	X	X	X
Datenwort			0		1	1	1		0	0	1	0	0	1	1
Prüfbits	0	1		1				1							
Codewort	0	1	0	1	1	1	1	1	0	0	1	0	0	1	1

Aufgabe 3 *Hamming-Distanz* (Punkte)
Berechnen Sie die Hamming-Distanz des Codes bestehend aus den folgenden Codewörter:

(a) 0000 0110 1011

Lösungsvorschlag: *(Punkte)*

Um die Hammingdistanz des Codes zu bestimmen, berechnet man die Hammingdistanz aller Codewörtern paarweise. Die paarweise geringste Distanz ist auch die Hammingdistanz des Codes.

Abbildung 4.1: Mit oben genannten Aufrufen generierte PDF-Datei, S.1

$$\begin{array}{cccc} 0 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 \end{array}$$

Die Hammingdistanz zwischen 0000 und 0110 beträgt 2.

$$\begin{array}{cccc} 0 & 0 & 0 & 0 \\ 1 & 0 & 1 & 1 \end{array}$$

Die Hammingdistanz zwischen 0000 und 1011 beträgt 3.

$$\begin{array}{cccc} 0 & 1 & 1 & 0 \\ 1 & 0 & 1 & 1 \end{array}$$

Die Hammingdistanz zwischen 0110 und 1011 beträgt 3.

Da die geringste Hammingdistanz zwischen den beiden Codewörtern 0000 und 0110 mit einem Wert von 2 vorliegt, beträgt die Hammingdistanz des gesamten Codes ebenfalls 2.

Aufgabe 4 *Gleitkommadarstellung* (Punkte)
Berechnen Sie für die folgende Dezimalzahl die binäre IEEE 754 Gleitkommadarstellung in Single Precision (32 Bit), wie Sie in der Vorlesung behandelt wurde. Geben Sie den Lösungsweg komplett an.

(a) -18.4

Lösungsvorschlag: *(Punkte)*

Die Mantisse ergibt sich aus der Zahl vor dem Komma als Binärzahl

$$18_{10} = 10010_2$$

und der Zahl nach dem Komma als Binärzahl. Für die Zahl nach dem Komma verwenden wir die Multiplikationsmethode:

$$\begin{array}{ll} 0.4 \cdot 2 = 0.8 & 0 \\ 0.8 \cdot 2 = 1.6 & 1 \\ 0.6 \cdot 2 = 1.2 & 1 \\ 0.2 \cdot 2 = 0.4 & 0 \\ 0.4 \cdot 2 = 0.8 & 0 \\ 0.8 \cdot 2 = 1.6 & 1 \\ 0.6 \cdot 2 = 1.2 & 1 \\ 0.2 \cdot 2 = 0.4 & 0 \\ 0.4 \cdot 2 = 0.8 & 0 \\ 0.8 \cdot 2 = 1.6 & 1 \\ \vdots & \\ \vdots & \\ \vdots & \end{array}$$

2

Abbildung 4.2: Mit oben genannten Aufrufen generierte PDF-Datei, S.2

$\Rightarrow 0,01100110011001100110011...$

Also: $18,4 = 10010,01100110011001100110011...$

Normalisiert erhält man: $1,00100110011001100110011... \cdot 2^4$

Da wir um 4 Stellen verschoben haben, erhalten wir für die Charakteristik des Exponenten: Charakteristik = Exponent + Bias = $4 + 127 = 131$ und $131_{10} = 10000011_2$

Das Vorzeichen ist negativ, also 1.

Vorzeichen	Exponent	Mantisse
1	10000011	00100110011001100110011

Aufgabe 5 *Rechnen im Zweierkomplement* (Punkte)
Addieren bzw. subtrahieren Sie die folgenden zwei Zahlen im Zweierkomplement.

(a) $55_{10} \qquad -15_{10}$

Lösungsvorschlag: *(Punkte)*

$$\begin{array}{cccccccccc} & 0 & 0 & 1 & 1 & & 0 & 1 & 1 & 1 \\ + & 1 & 1 & 1 & 1 & & 0 & 0 & 0 & 1 \\ \hline 1 & 1 & 1 & 1 & 0 & & 1 & 1 & 1 & \\ & 0 & 0 & 1 & 0 & & 1 & 0 & 0 & 0 \end{array}$$

Anmerkung: $55_{10} + -15_{10} = 40_{10}$

Abbildung 4.3: Mit oben genannten Aufrufen generierte PDF-Datei, S.3

Kapitel 5

Fazit

In diesem Kapitel wird ein kurzer Rückblick auf die Arbeit gegeben und das Ergebnis bewertet. Außerdem werden mögliche Erweiterungen aufgezeigt.

5.1 Bewertung der Arbeit

Der grundlegende Gedanke zu Beginn der Arbeit war, eine Anwendung zu entwickeln, mit der man typische Aufgaben aus der Rechnerarchitektur automatisch lösen und einen Lösungsweg erstellen kann. Wichtig beim Design der Anwendung war einfache Erweiterbarkeit ohne großen Aufwand und unkompliziertes Bedienen durch den Anwender. Diese Aspekte sind umgesetzt worden und die Anwendung kann in Zukunft viele, ansonsten händische, Arbeitsschritte erleichtern. Es ist möglich mit wenigen zusätzlichen Klassen neue Aufgaben hinzuzufügen.

Zudem wurde die Qualität der Algorithmen durch Unittests sichergestellt, sodass sich der Nutzer sicher sein kann, dass die Algorithmen das gewünschte Ergebnis liefern.

Die Anwendung ist als praktisches Hilfsmittel gedacht. Daher wurden Aspekte wie eine anschauliche Benutzeroberfläche vernachlässigt. Weiterhin gibt es die Möglichkeit zu Erweiterungen, auf die ich in Abschnitt 5.2 genauer eingehe.

5.2 Mögliche Erweiterungen

Die Erweiterungsmöglichkeiten für die Anwendung sind sehr zahlreich. Es gibt zum einen den Bereich der Benutzerfreundlichkeit und zum anderen den inhaltlichen Bereich.

Die Benutzeroberfläche ist im Rahmen meiner Arbeit sehr einfach gehalten worden. Aufgrund der Tatsache, dass ausgebildete Mitarbeiter des Fachbereichs Informatik die Anwendung nutzen, war dieser Aspekt nicht priorisiert. Eine mögliche Erweiterung ist daher eine anschaulichere Benutzeroberfläche, mit der es intuitiver möglich ist, die Anwendung zu nutzen.

Ein weiterer Aspekt sind verschiedene Optionen für die Ausführung. Der Ausgabepfad muss beispielsweise im Sourcecode angepasst werden. Diese Anpassung kann als optionaler Konsolenparameter eingeführt werden.

Die Anzahl an Aufgabentypen ist nach oben hin offen. Daher gibt es in dem Bereich viele Möglichkeiten zur Erweiterung. Beispiele für weitere Aufgabentypen sind:

- Caching mit verschiedenen Strategien
- KV-Diagramme
- logische Funktionen, Wahrheitstabellen, Normalformen, Minimalformen
- MAL-/Binär-Darstellung

Schließlich gibt es noch die Möglichkeit, andere Ausgabeformate zu implementieren. Die bisherige Ausgabe erfolgt in LaTeX. Denkbar sind Ausgabeformate wie HTML oder Markdown.

Literatur

[TA14] A.S. Tanenbaum und T. Austin. *Rechnerarchitektur: von der digitalen Logik zum Parallelrechner*. Always learning. Pearson, 2014. ISBN: 9783863266875. URL: https://books.google.de/books?id=0D-loAEACAAJ.

Ehrenwörtliche Erklärung

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

Düsseldorf, 25. November 2020 Markus Kaserer