

Teko schweizerische Fachhochschule

L-TIN-21-T-a

Projektarbeit: Brainfuck Compiler



Projektleiter: Mario Weilenmann, Marvin Huber

Dozent: Iwan Müller

Datum: 18.03.2023

Management Summary

Das Projektziel besteht darin, eine begleitende Projektdokumentation zu erstellen, die Implementierung der Brainfuck Programmiersprache durchzuführen und eine Visualisierung auf einem Embedded Hardware Tinkerforge (mit OLED Display) zu realisieren. Ein Teil des Projekts beinhaltet auch die Vertiefung der Kenntnisse in der Programmiersprache Java, indem ein Brainfuck-Compiler von Grund auf gebaut wird.

Das Projektteam setzt sich aus Mario Weilenmann und Marvin Huber zusammen. Mario wird sich auf die Entwicklung des Brainfuck-Compilers konzentrieren und dabei den Unterschied zwischen Compiler und Interpreter verstehen. Marvin wird sich auf den Aufbau unterschiedlicher Runtime-Umgebungen konzentrieren.

Der Kernaspekt des Projekts besteht darin, dass es den Aufbau der Programmierkenntnisse des Teams fördert. Durch die Zusammenarbeit und das gegenseitige Coaching bei der Entwicklung des Brainfuck-Compilers werden wir unser Verständnis für die Programmiersprache Java, die Brainfuck-Programmierung und die systematische Erstellung von Code vertiefen.

Das Projekt birgt auch Risiken, wie technische Schwierigkeiten, zeitliche Einschränkungen, mangelnde Zusammenarbeit, finanzielle Einschränkungen und fehlende Erfahrung. All diese Risiken könnten auftreten. Um diese Risiken zu minimieren, werden verschiedene Strategien eingesetzt, wie gründliche Vorbereitung und Forschung, realistische Zeitpläne, effektive Kommunikation und regelmässige Treffen sowie Schulung und Weiterbildung.

Das Projekt wurde erfolgreich abgeschlossen und wir sind mit dem Ergebnis zufrieden. Die Implementierung der Brainfuck-Programmiersprache und die Visualisierung auf dem Tinkerforge wurden erfolgreich umgesetzt und erfüllten unsere Erwartungen.

Insgesamt haben wir wertvolle Erfahrungen gesammelt und unsere Programmierkenntnisse erweitert. Wir empfehlen, die gewonnenen Erkenntnisse und Fähigkeiten in zukünftigen Projekten zu nutzen und weiterzuentwickeln.

Inhaltsverzeichnis

- [Teko schweizerische Fachhochschule](#)
 - [L-TIN-21-T-a](#)
 - [Projektarbeit: Brainfuck Compiler](#)
- [Management Summary](#)
- [Inhaltsverzeichnis](#)
- [Vorwort](#)
- [Initialisierungsphase](#)
 - [Ausgangslage](#)
 - [Projektauftrag](#)
 - [Zielsetzung](#)
 - [Ziele im Detail](#)
 - [Zusätzliche Ziele](#)
 - [Projektbeteiligte](#)
 - [Zusammenfassung](#)

- Abgrenzungen
- Risikoanalyse
 - Risikomanagement Tabelle
- Pflichtenheft
- Projektplan
- Meilensteine
- Projektstatusbericht
- Installationsanleitung
 - Technische Dokumentation
 - Raspberry PI Einrichtung
 - Raspberry Konfiguration
 - SSH einschalten
 - Statische IP vergeben
- Display HAT connector Setup
 - Brick Hat
 - Features
 - Beschreibung
 - Ressourcen
 - Erste Schritte
 - Interface
 - Installation Tinkerforge software
 - Brick Daemon (brickd)
 - Installation auf Raspberry PI OS
 - Einrichtung
 - Pakete
 - Übersicht Brick Viewer
 - Verwendung
 - Updates
 - Aktuellen Stand bestimmen / Nach Updates suchen
- Interpreter und Compiler
- Einführung
- Implementation mit Java
 - Einlesen des Brainfuck-Codes aus einer Datei
 - Speichern der Opcodes
 - Behandlung von Schleifen
 - Ausführen des Brainfuck-Codes
 - Behandlung von Fehlern
 - Klasse `BracketPair`
 - Enumeration `OpcodeEnum`
- Implementation
 - Aufbau des Projekts
 - Viewer
 - bf
 - Tokenizer:
 - Parser:
 - Optimizer:

- Pipeline:
 - Transpiler:
- Tokenizer
- Parser
- Pipeline
- Backends
- IR
- IR Code
- IR Erklärung
- Optimierungen
 - ConcatOptimizer
 - ConcatOptimizer Erklärung
 - ClearOptimizer
 - CopyOptimizer
- Tests
 - Rust Installation
 - Testcases
 - TestFile 1: Hello World
 - Input Online Compiler
 - Output online Compiler
 - Output Java
 - Output Rust
 - TestFile 2: bfeer
 - Input: ganzer Input in bfeerInput
 - Output Online Compiler
 - Output Java
 - Output Rust
 - TestFile 3: Mandelbrot
 - Output Online
 - Output Java
 - Output Rust
 - Unit Tests Optimizer
- Abschlussbericht
 - Erreichte Ziele
 - Schwierigkeiten
 - Verbesserungsvorschläge
 - Schlussfolgerung
- Schlussteil
 - Ausblick
 - Schlusswort
- Anhang
 - TestFiles

Vorwort

Wir haben uns für das Thema entschieden, einen Brainfuck-Compiler zu schreiben, da es für uns eine faszinierende Herausforderung darstellt. Brainfuck ist eine extrem simple, aber auch sehr abstrakte Programmiersprache, die lediglich aus acht Befehlen besteht. Aufgrund ihrer minimalistischen Natur ist es sehr schwierig, Brainfuck-Code zu lesen und zu schreiben.

Ein Brainfuck-Compiler ist ein Programm, das geschrieben wurde, um Brainfuck-Code in eine ausführbare Form zu übersetzen, die von einer Maschine verstanden werden kann. Ein Brainfuck-Compiler zu schreiben ist für uns aus folgenden Gründen interessant:

Erstens ist es eine grossartige Möglichkeit, um Programmierkenntnisse zu vertiefen und zu verbessern. Brainfuck ist eine sehr einfache Programmiersprache, die jedoch viele Konzepte der Informatik wie das Konzept des "Speichers" oder "Pointer" erfordert, um sie effektiv zu nutzen. Durch das Schreiben eines Compilers für diese Sprache kann man das Verständnis für diese Konzepte vertiefen und gleichzeitig das eigene Programmiervermögen verbessern.

Zweitens kann es für die Forschung in der theoretischen Informatik von Nutzen sein. Brainfuck wird oft als Beispiel für eine sogenannte "Turing-vollständige" Sprache verwendet, was bedeutet, dass sie in der Lage ist, jede Berechnung durchzuführen, die auch von einem Universalcomputer durchgeführt werden kann. Dies bedeutet, dass das Schreiben eines Compilers für Brainfuck ein interessantes Beispiel für die Erforschung von Berechnungen und Programmiersprachen sein kann.

Drittens kann es einfach aus Gründen der Unterhaltung sein. Wir haben ein grosses Interesse an spannend informatik Themen, finden es einfach faszinierend, mit Brainfuck zu arbeiten und zu sehen, wie sich komplexe Algorithmen mit nur wenigen Befehlen schreiben lassen.

Es bietet die Möglichkeit, Programmierkenntnisse zu verbessern, theoretische Konzepte zu erforschen und sich einfach mit einer faszinierenden Programmiersprache zu beschäftigen. Insgesamt ist das Schreiben eines Brainfuck-Compilers eine herausfordernde und interessante Aufgabe, die wir uns Stellen möchten aufgrund von den oberen genannten Gründen, deswegen haben wir uns entschieden dafür.

Initialisierungsphase

Ausgangslage

Die Ausgangslage für die Projektarbeit besteht darin, dass wir über limitierte Java-Kenntnisse verfügen und diese gerne erweitern möchten. Dies ist ein wichtiger Aspekt, da die Programmiersprache Java eine der am häufigsten verwendeten Programmiersprachen ist und in vielen Bereichen der Softwareentwicklung eingesetzt wird.

Doch nicht nur das Erlernen von Java ist ein Ziel der Projektarbeit. Vielmehr möchten wir verstehen, wie ein Compiler funktioniert. Ein Compiler ist ein Programm, das Quellcode einer Programmiersprache in Maschinencode übersetzt, damit dieser vom Computer ausgeführt werden kann. Das Verständnis der Funktionsweise eines Compilers ist daher von entscheidender Bedeutung für die Entwicklung von Software und die Verbesserung von Programmen.

Um dieses Ziel zu erreichen, haben wir uns dazu entschieden, simplen Programmcode der Programmiersprache Brainfuck zu schreiben und zu kompilieren. Brainfuck ist eine Turing-vollständige Programmiersprache, die nur aus acht Befehlen besteht und daher als einfach zu erlernen gilt. Doch obwohl

Brainfuck eine einfache Sprache ist, ist sie aufgrund ihrer geringen Abstraktionsebene und der beschränkten Anzahl von Befehlen eine Herausforderung für jeden Compiler.

Das Wissen und die Fähigkeiten, die während der Projektarbeit erworben werden, sind von unschätzbarem Wert für zukünftige Projekte und Karrieremöglichkeiten in der Softwareentwicklung und hat grosses potential in der Richtung Java.

Projektauftrag

Projektauftrag: Erstellung einer Projektdokumentation, Implementierung von Brainfuck Programmiersprache und Visualisierung an Embedded Hardware Tinkerforge.

Zielsetzung

Dieses Projekt zielt darauf ab, eine begleitende Projektdokumentation zu erstellen, die die Entwicklung und Implementierung eines Brainfuck Compilers umfasst. Das Ziel besteht darin, die Programmiersprache Brainfuck zu implementieren und diese auf einer Embedded Hardware Tinkerforge (OLED Display) zu visualisieren. Ein Teil des Projekts beinhaltet auch, dass wir uns gegenseitig beraten und unterstützen, um unser Wissen in Java, Brainfuck und systematischen Abläufen beim Erstellen von Codes zu vertiefen.

Ziele im Detail

1. Erstellung einer begleitenden Projektdokumentation: Es wird eine detaillierte Projektdokumentation erstellt, die den Entwicklungsprozess, die Implementierung und die Ergebnisse des Projekts beschreibt. Die Dokumentation umfasst auch Anweisungen und Anleitungen für die Implementierung des Brainfuck Compilers und die Visualisierung der Programmiersprache auf der Embedded Hardware Tinkerforge.
2. Implementierung von Brainfuck Programmiersprache: Ein wesentliches Ziel des Projekts besteht darin, die Programmiersprache Brainfuck zu implementieren. Dies beinhaltet die Entwicklung eines Brainfuck Compilers von Grund auf, um sicherzustellen, dass das Verständnis für Java und Brainfuck vertieft wird. Wir werden auch sicherstellen, dass wir die Unterschiede zwischen Compiler und Interpreter verstehen und in der Lage sind, diese zu erklären.
3. Visualisierung an Embedded Hardware Tinkerforge: Ein weiteres Ziel des Projekts ist es, die Brainfuck Programmiersprache auf einer Embedded Hardware wie Tinkerforge zu visualisieren. Dies beinhaltet die Entwicklung einer Runtime-Umgebung, um sicherzustellen, dass die Programmiersprache auf der Hardware effektiv und effizient dargestellt wird. Marvin wird sich auf die Entwicklung von verschiedenen Runtime-Umgebungen konzentrieren, um sicherzustellen, dass die Visualisierung der Programmiersprache auf der Hardware optimal abläuft.

Zusätzliche Ziele

1. Vertiefung des Verständnisses von Java: Die Entwicklung des Brainfuck Compilers von Grund auf bietet eine hervorragende Gelegenheit, unser Verständnis von Java zu vertiefen. Wir werden sicherstellen, dass wir ein tiefes Verständnis der Programmiersprache Java haben für unseren Zweck und in der Lage sind, effektiv damit zu arbeiten.
2. Vertiefung des Verständnisses von Brainfuck: Mit der Implementierung können wir unser Verständnis von Brainfuck vertiefen. Dies ist notwendig, um dem Compiler zu schreiben.

3. Entwicklung von systematischen Abläufen: Ein wichtiger Teil des Projekts ist die Entwicklung von systematischen Abläufen beim Erstellen von Codes. Wir werden sicherstellen, dass wir die besten Praktiken bei der Entwicklung von Codes einsetzen, um die Qualität unserer Arbeit zu verbessern.
4. Verbesserung der Zusammenarbeit: Ein weiteres Ziel des Projekts besteht darin, unsere Zusammenarbeit zu verbessern. Wir werden uns gegenseitig beraten und unterstützen, um sicherzustellen, dass jeder von uns das Projekt erfolgreich abschliessen kann. Wir werden auch sicherstellen, dass wir effektive Kommunikationswege benutzen, um sicherzustellen, dass wir uns während des gesamten Projekts auf dem gleichen Stand halten und effizient arbeiten können.
5. Verbesserung der technischen Fähigkeiten: Dieses Projekt bietet uns eine gute Gelegenheit, unsere technischen Fähigkeiten zu verbessern. Wir werden viel mit Brainfuck, Java und Tinkerforge arbeiten und uns tief in die Materie einarbeiten müssen. Wir werden dabei viel neues dazulernen.

Projektbeteiligte

Mario Weilenmann: Mario ist verantwortlich für die Entwicklung des Brainfuck Compilers und die Implementierung der Programmiersprache auf der Embedded Hardware Tinkerforge (OLED Display).

Marvin Huber: Marvin wird sich auf die Entwicklung von verschiedenen Runtime-Umgebungen konzentrieren, um sicherzustellen, dass die Visualisierung der Programmiersprache auf der Embedded Hardware Tinkerforge (OLED Display) optimal ist. Er wird auch Mario bei der Entwicklung des Brainfuck Compilers unterstützen und sicherstellen, dass der Code effektiv und effizient implementiert wird.

Zusammenfassung

Insgesamt zielt dieses Projekt darauf ab, eine begleitende Projektdokumentation zu erstellen, die die Entwicklung und Implementierung eines Brainfuck Compilers umfasst. Wir werden sicherstellen, dass wir ein tiefes Verständnis von Java, Brainfuck und systematischen Abläufen beim Erstellen von Codes haben. Wir werden auch sicherstellen, dass die Brainfuck Programmiersprache auf der Hardware Tinkerforge visualisiert wird. Damit können wir unsere Programmierfähigkeiten verbessern und lernen, besser zusammen zu arbeiten. Wir sind zuversichtlich, dass wir das Projekt erfolgreich abschliessen werden und die gesteckten Ziele erreichen werden.

Abgrenzungen

Das Projekt, das von uns durchgeführt wird, hat klare Abgrenzungen hinsichtlich seines Umfangs und seiner Zielsetzungen. Eines dieser Ziele ist die Erlernung grundlegender Programmier-Methodiken. Dies bezieht sich auf die Konzepte und Techniken, die in der Programmierung verwendet werden, um effektiv und effizient zu arbeiten.

Wir werden in diesem Projekt grundlegende Programmierkonzepte wie Variablen, Schleifen, Bedingungen und Funktionen kennenlernen sowie der Unterschied zwischen Compiler und Interpreter und auf verschiedenen Runtime umgebungen arbeiten. Diese Konzepte sind von grundlegender Bedeutung für jede Programmiersprache und bilden die Grundlage für komplexe Programme. Indem wir diese Konzepte erlernen und anwenden, können wir unsere Fähigkeiten im Bereich der Programmierung verbessern.

Es ist jedoch wichtig zu betonen, dass das Projekt nicht den Anspruch hat, zur Produktionsreife zu gelangen. Das bedeutet, dass das Ziel des Projekts nicht darin besteht, ein voll funktionsfähiges Produkt zu entwickeln,

das von anderen genutzt werden kann. Stattdessen soll uns das Projekt die Möglichkeit geben, grundlegende Konzepte der Programmierung zu erlernen und anzuwenden und uns iJava-programming voran zu bringen.

Risikoanalyse

Eine Risikoanalyse ist eine wichtige Komponente bei der Planung eines Projekts, um potenzielle Risiken zu identifizieren und Strategien zu entwickeln, um diese Risiken zu minimieren oder zu vermeiden. Im Folgenden sind einige Risiken aufgeführt, die bei der Entwicklung des Brainfuck Compilers und der Visualisierung der Programmiersprache auf der Embedded Hardware Tinkerforge (OLED Display) auftreten könnten:

1. Technische Schwierigkeiten: Das Projekt erfordert ein gewisses Verständnis von Java, Brainfuck und Tinkerforge. Es besteht das Risiko, dass technische Schwierigkeiten auftreten könnten, die das Projekt verzögern oder sogar unmöglich machen können. Um dieses Risiko zu minimieren, werden wir sicherstellen, dass wir eine gründliche Recherche betreiben und Vorbereitung durchführen und dass wir in der Lage sind, die technischen Herausforderungen effektiv zu bewältigen.
2. Zeitliche Einschränkungen: Das Projekt hat einen relativ kleinen Zeitrahmen, innerhalb dessen das Projekt abgeschlossen werden muss. Es besteht das Risiko, dass wir aufgrund von unvorhergesehenen Schwierigkeiten oder Komplikationen den Zeitrahmen nicht einhalten können. Um dieses Risiko zu minimieren, werden wir sicherstellen, dass wir realistische Zeitpläne erstellen und dass wir unser Bestes tun, um innerhalb dieser Zeitpläne zu arbeiten.
3. Mangelnde Zusammenarbeit: Das Projekt erfordert eine enge Zusammenarbeit zwischen den Projektbeteiligten. Es besteht das Risiko, dass wir nicht effektiv zusammenarbeiten können, was zu Verzögerungen oder Problemen im Projekt führen kann. Um dieses Risiko zu minimieren, werden wir sicherstellen, dass wir effektive Kommunikationswege etablieren und dass wir uns regelmässig treffen, (fast jedes Wochenende) um den Fortschritt des Projekts zu besprechen.
4. Finanzielle Einschränkungen: Das Projekt erfordert finanzielle Ressourcen für die Beschaffung von Materialien. Es besteht das Risiko, dass das Projekt aufgrund von finanziellen Einschränkungen gestoppt oder verzögert wird. Um dieses Risiko zu minimieren, werden wir sicherstellen, dass wir realistische Budgets erstellen und dass wir alternative Finanzierungsoptionen prüfen, falls nötig.
5. Fehlende Erfahrung: Es besteht das Risiko, dass wir möglicherweise nicht über ausreichende Erfahrung in der Programmierung oder Implementierung von Codes verfügen, um das Projekt erfolgreich abzuschliessen. Um dieses Risiko zu minimieren, werden wir sicherstellen, dass wir uns gegenseitig unterstützen und dass wir mit Selbststudium und Recherchieren an die gewünschten Informationen kommen.

Zusammenfassend kann gesagt werden, dass das Projekt einige Risiken birgt. Wir werden jedoch sicherstellen, dass wir diese Risiken im Auge behalten und dass wir Strategien entwickeln, um diese Risiken zu minimieren oder zu vermeiden. Wir sind zuversichtlich, dass wir das Projekt erfolgreich abschliessen werden und dass wir die gesteckten Meilensteine erreichen werden.

Risikomanagement Tabelle

Risiko	Eintreffwahrscheinlichkeit	Schweregrad	Strategie zur Risikominimierung
--------	----------------------------	-------------	---------------------------------

Risiko	Eintreffwahrscheinlichkeit	Schweregrad	Strategie zur Risikominimierung
Technische Schwierigkeiten	Mittel	Leicht-Mittel	Gründliche Vorbereitung und Forschung, technische Expertise nutzen, um Herausforderungen effektiv zu bewältigen
Zeitliche Einschränkungen	Hoch	Leicht	Realistische Zeitpläne erstellen, Priorisierung der Aufgaben, um sicherzustellen, dass das Projekt innerhalb des Zeitrahmens abgeschlossen wird
Mangelnde Zusammenarbeit	Leicht	Mittel	Effektive Kommunikation und regelmässige Treffen, um sicherzustellen, dass das Team zusammenarbeitet und Schwierigkeiten frühzeitig erkannt und gelöst werden
Finanzielle Einschränkungen	Niedrig	Mittel	Realistische Budgets erstellen, alternative Finanzierungsmöglichkeiten prüfen, falls nötig
Fehlende Erfahrung	Hoch	Leicht-Mittel	Selbststudium, gegenseitige Unterstützung und Mentoring, um sicherzustellen, dass das Team über ausreichende Fähigkeiten verfügt. Profitierung von Marvins kenntnissen.

Pflichtenheft

Mario	Marvin
Erstellen der Tabelle und Meilenstein Daten	Vorbereitung des Programmkonzeptes
Aufsetzung RASPI	Überarbeitung des Programmcodes in Rust
Einlesen Brainfuck	Rust Vorbereitung für RPI
Enums in Java vorbereiten	
Filereader Selbststudium: Print file content as text	
Zeichen Brainfuck Erklärung verstehen	
Raspberry PI mit Brick hat fertigstellen	

Mario	Marvin
Memory Pointer Selbststudium	
Byte Arrays Verständnis aufbauen	Workshop erstellen mit Erklärung für Mario
Bracket Implementation verstehen	Vorbereitung der Erklärung: Stacks / Pair Funktion / Logik der Bracketpairs
Dokumentation Java Code	Dokumentation Brainfuck und Rust

Projektplan

1. Konzeption (Benötigte Komponenten, Aufbau, Vorgehen)
2. Versuchsaufbau mit Tinkerforge (Red Brick, OLED Display)
3. Programmierung von Interpreter und Compiler (Auslesung und Speicherung der Daten)
4. Entwicklung der Darstellung von Output
5. (Visualisierung der Daten)
6. Finalisierung der Dokumentation

Meilensteine

Meilensteine	Ziel der Fertigstellung	Fertiggestellt am
Konzeption (Benötigte Komponenten, Aufbau, Vorgehen)	bis 11.12.2022	11.12.2022
Versuchsaufbau mit Tinkerforge (Red Brick, OLED Display)	bis 01.03.2023	19.02.2023
Programmierung von Interpreter und Compiler	bis 18.02.2023	04.03.2023
Entwicklung der Darstellung von Output	bis 25.02.2023	04.03.2023
Visualisierung der Daten	bis 25.02.2023	04.03.2023
Finalisierung der Dokumentation	bis 12.03.2023	12.03.2023

Projektstatusbericht

Datum	Ausgeübte Tätigkeiten	Wer
4.12.2022	<ul style="list-style-type: none"> - Erstellen des Repositorys, Besprechung der Richtlinien und Anlegen der Dateien. - Überarbeitung des Projektantrages. - Besprechung und Bestellung Hardware 	MV, MA
10.12.2022	<ul style="list-style-type: none"> - Absprache von Codeidee, erste Entwürfe und Codespezifizierungen 	MV, MA
17.12.2022	<ul style="list-style-type: none"> - Start Einlesen der Programmiersprache - Absprache der Funktionalität der Programmiersprache 	MV, MA

Datum	Ausgeübte Tätigkeiten	Wer
31.12.2022	- Start der Entwicklung und Konzeptbesprechung der jeweiligen Programmieraufgaben - Entwicklungsstart und Einlesen von Enums	MV, MA
8.01.2023	- Entwicklung der Funktionalität des FileReaders	MA
14.01.2023	- Entwicklung der Funktionalität der Zeichen-Semantik	MV, MA
21.01.2023	- Raspberry PI Setup und Testing des Brick-hats	MA
29.01.2023	- Entwicklung des Memory Pointers - Workshop Byte Arrays	MV, MA
4.02.2023	- Entwicklung der Stackfunktion - Theorieblock Bracketlist	MV, MA
18.02.2023	- Implementation Bracketlist - Implementation Bracketpair	MV, MA
5.03.2023	- Errorhandling der Bracketpair Funktion - Entwicklung Bracket.get und Indexing Funktion	MV, MA
11.03.2023	- Code Cleanup, Finishing Touch - Dokumentation Raspberry PI Setup - Layout Anpassungen - Dokumentation Code	MV, MA
12.03.2023	- Dokumentationserweiterungen	MV, MA

Installationsanleitung

Technische Dokumentation

Raspberry PI Einrichtung

Für die Visuelle demo benötigen wir einen Host. Wir haben uns hier für ein Raspberry PI entschieden, da dieses einfach zu warten ist und wenig Platz benötigt.

Mit Hilfe von Raspberry PI Imager kann das gewünschte Betriebssystem installiert werden. Wir verwenden das PI OS 64-bit und schreiben dieses auf die Micro SD Karte. Bei erfolgreichem schreiben des Betriebssystems können wir die SD Karte einsetzen und mit einem USB-C den Raspberry Pi Computer mit Strom versorgen.

Raspberry Konfiguration

Mittels HDMI können wir den Desktop des Betriebssystems anzeigen lassen und die ersten Konfigurationen vornehmen.

Mittels Setupmanager wählen wir als erstes die korrekte Region, Zeitzone und Sprache. Danach setzen wir ein Passwort sowie den Namen für den User. Wir benennen ihn nach unserem Projekt, also sugu. Wir benötigen den Namen und Passwort später, wenn wir via SSH darauf zugreifen wollen.

Wir setzen den Haken für das Screen Setup um optimierte Auflösungen zu erhalten für den Bildschirm. Da wir Ethernet verwenden, müssen wir uns nicht mit einem W-LAN verbinden.

Sobald wir mit einem Netzwerk verbunden sind, können wir im Setupmanager ein Update der Software durchführen.

SSH einschalten

Um auf unseren User zugreifen zu können, müssen wir SSH aktivieren. Dafür klicken wir auf dem Desktop auf das Raspberry logo, wählen **Preferences** und dann **Raspberry PI Configuration**. Unter dem **Interfaces** Tab können wir SSH anwählen.

Statische IP vergeben

Um mit keinen Komplikationen konfrontiert zu werden, vergeben wir dem Raspberry PI eine Statische IP Adresse. Dafür öffnen wir das Terminal auf dem Desktop. Mittels `ifconfig` können wir die momentan zugewiesene IP des netzwerkes ansehen. In unserem Fall ist dies **192.168.1.17**.

Mittels `sudo nano /etc/dhcpd.conf` können wir direkt in die config file unsere gewünschte IP Adresse schreiben. Dies machen wir wiefolgt:

```
interface eth0
static ip_address=192.168.1.17
static routers=192.168.1.1
static domain_name_servers=8.8.8.8 8.8.4.4
```

Einstellungen mit `Ctrl + o` schreiben und den Editor mit `Ctrl + x` verlassen. Danach den Raspi neustarten:

`sudo reboot`

Display HAT connector Setup

Brick Hat

Features

- Raspberry Pi HAT im Standard-HAT-Formfaktor
- **Acht** Anschlüsse für Bricklets
- Integrierte 5,3V Stromversorgung (6V-28V Eingang, bis zu 4A)
- Misst USB- und DC-Spannungsversorgung
- Bietet eine Real-Time Clock für den Raspberry Pi
- Bietet Schlafmodus (Low Power) und Watchdog

Beschreibung

Der HAT Brick ist ein [Raspberry Pi HAT](#) im Standard-HAT-Formfaktor. Der Brick ist zur HAT-Spezifikation konform und funktioniert automatisch und ohne Änderungen mit Raspbian.

Mit dem HAT Brick können bis zu **acht Bricklets** an ein Raspberry Pi angeschlossen werden.

Note: Der HAT Brick besitzt 7-Pol-Bricklet-Anschlüsse. Über ein 7-Pol- <-> 7-Pol-Kabel können Bricklets an den Brick angeschlossen werden. Es werden nur Bricklets unterstützt, die über einen 7-poligen Anschluss verfügen.

Der Raspberry Pi kann über den HAT Brick mit einer externen 6V-28V DC Stromversorgung betrieben werden. Die integrierte Stromversorgung liefert auch unter grosser Last stabile 5V für den Raspberry Pi. Somit können auch angeschlossene Bricklets und verbundene USB-Geräte versorgt werden. Das HAT Brick liefert hierfür eine etwas erhöhte Spannung von 5,3V.

Alternativ können HAT Brick und Raspberry Pi auch über USB-C versorgt werden. In diesem Fall muss allerdings sichergestellt werden, dass die Stromversorgung stabile 5V bietet. Dies ist zum Beispiel mit dem offiziellen Raspberry Pi Universal-Netzteil möglich. Die USB/DC Versorgungsspannungen werden vom HAT gemessen und sind über die API zugänglich.

Der HAT Brick bietet eine [Real-Time Clock mit Super-Cap Backup](#), die direkt mit dem Raspberry Pi verbunden ist. Mit dieser kann [der Raspberry Pi für eine angegebene Zeit ausgeschaltet werden](#). Somit kann der Raspberry Pi auch in batteriebetriebenen Anwendungen eingesetzt werden, zum Beispiel in einer Anwendung, in der Sensorinformationen jede Stunde in die Cloud geschickt werden sollen.

Mit dem HAT Brick kann ein [Watchdog](#) implementiert werden, der den Raspberry Pi neustartet, wenn sich dieser aufhängt oder das eigene Programm steckenbleibt.

Der HAT Brick ist elektronisch kompatibel zu den Raspberry Pis 2B, 3B, 3B+, 4B, Zero und Zero W. Die Befestigungslöcher sind kompatibel zum Raspberry Pi 2/3/4. Zusätzlich bieten wir mit dem [HAT Zero Brick](#) eine kleinere Version, deren Befestigungslöcher zum Raspberry Pi Zero kompatibel sind.

Eigenschaft	Wert
Stromverbrauch	100mW (20mA bei 5V)
Bricklet-Anschlüsse	8
DC Eingangsspannungsbereich	6V-28V
DC Ausgang	5,3V, max. 4A
Stromverbrauch im Sleepmodus (≤ 1.4)*	70mW (14mA bei 5V) + 1.5mW wenn die Sleep-LED aktiv ist
Abmessungen (B x T x H)	65 x 56 x 25mm (2,56 x 2,20 x 0,98")
Gewicht	30g

Ressourcen

- Schaltplan ([Download](#))
- Umriss und Bohrplan ([Download](#))

- Quelltexte und Platinenlayout ([Download](#))
- 3D Modell ([Online ansehen](#) | Download: [STEP](#), [FreeCAD](#))

Erste Schritte

Um den HAT Brick verwenden zu können, muss zuerst der [Brick Daemon](#) auf dem Raspberry Pi installiert werden. Der Brick Daemon agiert als Proxy zwischen den Brickletanschlüssen des HAT Brickss und den API Bindings. Er kümmert sich auch um die Real-Time Clock.

Interface

Es wird ein Interface verwendet mit dem Namen "Brick Viewer" für eine visuelle veranschaulichung. Der Brick Viewer kann entweder direkt auf dem Raspberry Pi oder auf einem externen PC, der über Ethernet oder WLAN Zugriff auf den Raspberry Pi besitzt, installiert werden. Von einem externen PC aus muss sich auf den Hostnamen oder die IP des Raspberry Pis verbunden werden, vom Raspberry Pi aus auf localhost. Wir haben uns entschieden, die Installation direkt auf dem Raspberry Pi durchzuführen, um aus Gründen der Einfachheit bei einem Gerät zu bleiben.

Installation Tinkerforge software

Brick Daemon (brickd)

Der Brick Daemon ist ein Daemon (bzw. Service für Windows) der als eine Brücke zwischen [Bricks/Bricklets](#) und den [API Bindings](#) für die verschiedenen Programmiersprachen fungiert.

Der Daemon leitet Daten zwischen der USB Verbindung und den TCP/IP Sockets hin und her. Bei der Benutzung der API Bindings wird eine TCP/IP Verbindung zum Brick Daemon hergestellt. Dieses Konzept erlaubt es Bindings für nahezu jede Programmiersprache ohne Abhängigkeiten zu erstellen. Dadurch ist es möglich Bricks und Bricklets über eingebettete Geräte wie Smartphones zu programmieren, die nur spezifische Programmiersprachen unterstützten.

Zusätzlich ist es möglich den PC auf dem der Brick Daemon läuft von dem PC auf dem der Benutzercode läuft zu trennen. Dadurch ist das Steuern über ein Smartphone oder auch über das Internet möglich.

Installation auf Raspberry PI OS

Die auf der [Download Seite](#) verfügbare Software steht auch als Debian Pakete in unserem [APT Repository](#) zur Installation bereit.

Aktuell werden folgende Distributionen und Architekturen unterstützt:

- [Debian](#): amd64, i386, armhf, arm64
- [Ubuntu](#): amd64, i386, armhf, arm64
- [Raspberry Pi OS](#) (ehemals Raspbian): armhf

Einrichtung

Schritt 1: Öffentlichen GPG Schlüssel importieren:

```
wget https://download.tinkerforge.com/apt/$(. /etc/os-release; echo $ID)/tinkerforge.gpg  
-q -O - | sudo tee /etc/apt/trusted.gpg.d/tinkerforge.gpg > /dev/null
```

Schritt 2: APT Repository hinzufügen:

```
echo "deb https://download.tinkerforge.com/apt/$(. /etc/os-release; echo $ID  
$VERSION_CODENAME) main" | sudo tee /etc/apt/sources.list.d/tinkerforge.list
```

Schritt 3: APT Paketliste aktualisieren:

```
sudo apt update
```

Pakete

Aktuell sind folgende Pakete verfügbar:

- Tools
 - **Brick Daemon:** `brickd`
 - **Brick Viewer:** `brickv`
 - **Brick Flash:** `brick-flash`
- API Bindings
 - **Go:** `golang-tinkerforge-dev`
 - **Java:** `libtinkerforge-java` und `libtinkerforge-java-doc`
 - **JavaScript (Node.js):** `node-tinkerforge`
 - **JSON:** `tinkerforge-json`
 - **MQTT:** `tinkerforge-mqtt`
 - **Octave:** `octave-tinkerforge`
 - **Perl:** `libtinkerforge-perl`
 - **PHP:** `php-tinkerforge`
 - **Python:** `python3-tinkerforge` (Python 3) und `python-tinkerforge` (Python 2)
 - **Ruby:** `ruby-tinkerforge`
 - **Shell:** `tinkerforge-shell`

Übersicht Brick Viewer

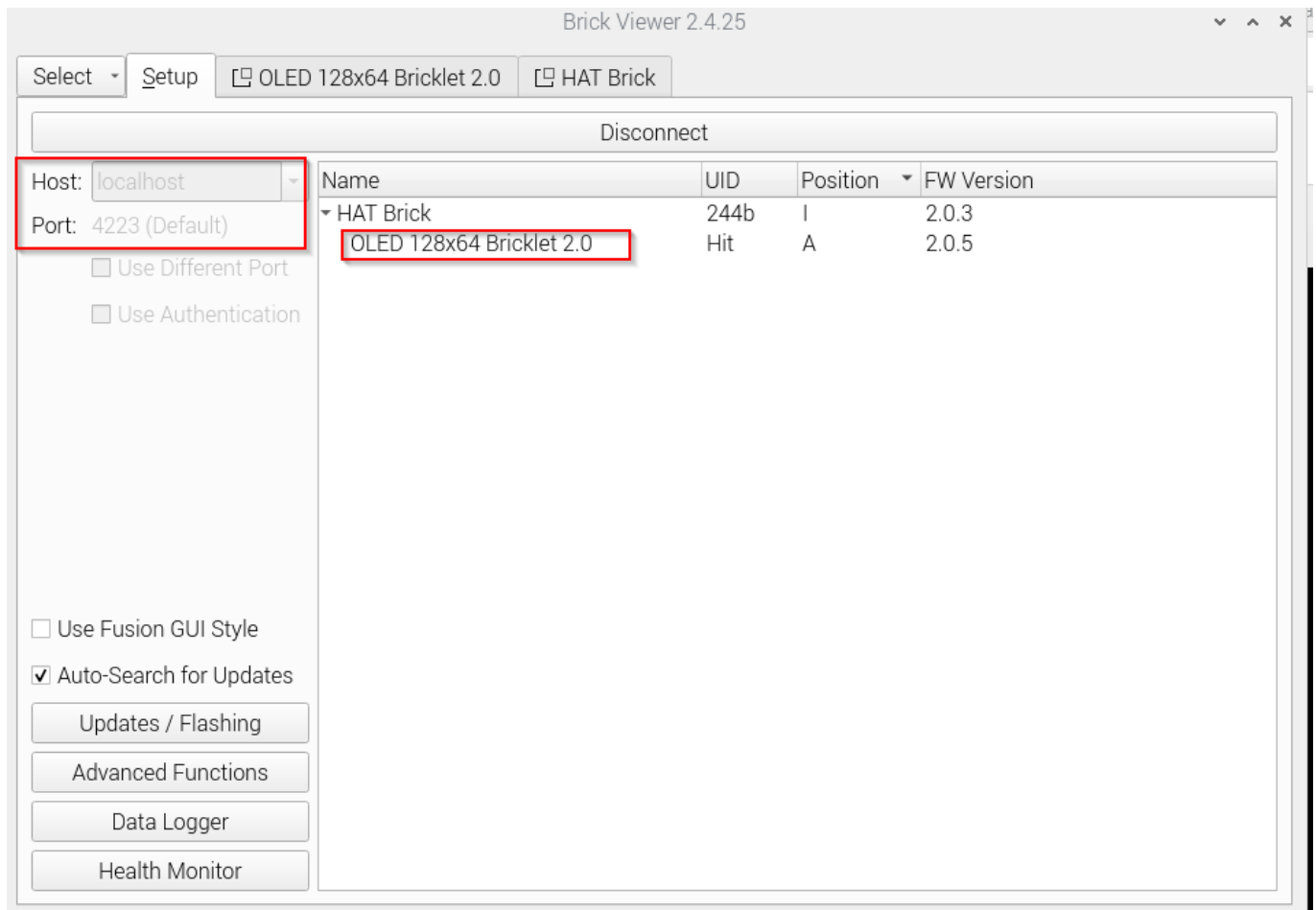
Der Brick Viewer bietet eine graphische Oberfläche um **Bricks** und **Bricklets** zu testen. Jedes Gerät hat seine eigenen Tab der die Hauptfunktionen abbildet und erlaubt diese zu steuern.

Darüber hinaus kann der Brick Viewer verwendet werden, um den Analog-Digital-Wandler der Bricks zu **kalibrieren** und so deren Messqualität zu verbessern, und um **Brick Firmwares** und **Bricklet Plugins** zu flashen.

Verwendung

Zuerst muss der Brick Viewer mit dem **Brick Daemon** oder z.B. einer **WIFI Extension** verbunden werden. Dabei kann der Brick Daemon auf dem gleichen oder einem anderen PC als der Brick Viewer laufen. Dazu zuerst die IP Adresse des PCs auf dem der Brick Daemons läuft oder die IP Adresse einer WIFI Extension als Host angeben. Falls Brick Daemon und Viewer auf dem gleichen PC laufen kann der

Standardwert `localhost` beibehalten werden. Nach einem Klick auf den "Connect" Knopf werden die verbundenen Bricks und Bricklets auf je einem eigenen Tab angezeigt und können getestet werden. Da wir beides auf dem Raspberry PI installiert haben, verbinden wir Brick Viewer über Localhost.



Nun ist die Verbindung zum Display hergestellt und wir können weiterfahren.

Updates

Ein Klick auf den "Updates / Flashing" öffnen einen Dialog mit Informationen über verfügbare Updates und der Möglichkeit Bricks und Bricklets neu zu flashen. Der "Advanced Functions" Knopf öffnet einen Dialog zur Kalibrierung des Analog-Digital-Wandler eines Bricks.

Aktuellen Stand bestimmen / Nach Updates suchen

Nach dem Start des Brick Viewers und dem Verbinden zu einem Brick Daemon oder einer Master Extension kann überprüft werden ob eine neuere Software für die angeschlossenen Geräte verfügbar ist.

Hierzu muss auf "Updates / Flashing" geklickt werden. Der Dialog zeigt die angeschlossenen Geräte und deren Software stand. Orange unterlegte Einträge können geupdatet werden. Rot unterlegte Einträge müssen geupdatet werden damit sie korrekt funktionieren.

Der Dialog ermöglicht es alle Bricklets gleichzeitig über den Knopf "Auto-Update All Bricklets" auf die neuste Softwareversion zu bringen. Bricks können nicht automatisch auf den neusten Stand gebracht werden (siehe [Brick Firmware Flashing](#)).

Interpreter und Compiler

Ein Compiler und ein Interpreter sind zwei Arten von Programmen, die für die Ausführung von Code verwendet werden können. Beide sind für die Umwandlung von Programmiersprache in ausführbaren Code verantwortlich, jedoch auf unterschiedliche Weise.

Ein Compiler übersetzt den gesamten Quellcode eines Programms in eine ausführbare Datei, die von einem Computer oder Gerät direkt ausgeführt werden kann. Der Compiler prüft dabei den Quellcode auf Syntax- und Semantikfehler, wandelt den Code in eine Zwischensprache um und optimiert den Code für die Zielplattform. Das Ergebnis ist eine Datei, die schnell und effizient ausgeführt werden kann. Beispiele für Compiler sind der GCC-Compiler für C und C++, der Java-Compiler für Java und der Swift-Compiler für Swift.

Ein Interpreter hingegen liest und interpretiert den Quellcode zur Laufzeit, Zeile für Zeile. Der Interpreter wandelt dabei den Quellcode in ausführbaren Code um, ohne eine ausführbare Datei zu erstellen. Der Interpreter prüft dabei ebenfalls auf Syntax- und Semantikfehler und führt den Code sofort aus. Beispiele für Interpreter sind der Python-Interpreter für Python, der Ruby-Interpreter für Ruby und der JavaScript-Interpreter für JavaScript.

Der grösste Unterschied zwischen einem Compiler und einem Interpreter besteht darin, wie sie den Code ausführen. Ein Compiler übersetzt den gesamten Quellcode in ausführbaren Code, bevor das Programm ausgeführt wird, während ein Interpreter den Code Zeile für Zeile interpretiert und ausführt. Ein Compiler erfordert somit mehr Zeit für die Übersetzung, bevor das Programm ausgeführt werden kann, während ein Interpreter sofort ausführen kann, sobald der Code eingegeben wurde. Dies führt zu unterschiedlicher Leistung und Effizienz, wobei Compiler in der Regel schneller und effizienter sind als Interpreter.

Einführung

Was ist brainfuck überhaupt? Brainfuck ist eine esoterische Programmiersprache, die extrem minimalistisch ist und nur aus acht Befehlen besteht. Die Sprache wurde von Urban Müller im Jahr 1993 entwickelt und ist dafür bekannt, dass sie sehr schwer zu lesen und zu schreiben ist. Obwohl Brainfuck nur aus wenigen Befehlen besteht, können dennoch komplexe Programme geschrieben werden. Die Sprache erfordert jedoch viel Übung und Geduld, um effektiv verwendet werden zu können.

Die Befehle:

Befehl	Bedeutung
>	Verschiebe den Datenzeiger um eine Zelle nach rechts
<	Verschiebe den Datenzeiger um eine Zelle nach links
+	Erhöhe den Wert der aktuellen Zelle um eins
-	Verringere den Wert der aktuellen Zelle um eins
.	Gib den Wert der aktuellen Zelle als ASCII-Code aus
,	Nimm eine Eingabe entgegen und speichere sie in der aktuellen Zelle
[Beginne eine Schleife. Führe die Schleife aus, solange der Wert der aktuellen Zelle nicht null ist.

Diese Befehle können in verschiedenen Kombinationen verwendet werden, um komplexere Programme zu schreiben. Es ist jedoch wichtig zu beachten, dass Brainfuck aufgrund seiner minimalistischen Natur sehr schwer lesbar und schreibbar ist. Es erfordert viel Übung und Geduld, um effektiv in dieser Sprache zu programmieren.

Ein kleines Beispiel von Hello World! Programm in Brainfuck:

```
+++++++ [ >++++++>++++++++>++++>+<<<<- ]>+.,>+.+++++. .+++.>+.,
<<+++++++++.>+.+++.-----.-----.>+.,>.
```

Jetzt die Erklärung was das Programm macht Step bz Step.

Brainfuck-Code	Aktion
+++++ +++++	Initialisiert den Zähler (Speicherzelle #0) mit dem Wert 10
[Beginn einer Schleife, um die nächsten vier Speicherzellen auf 70/100/30/10 zu setzen
>+++++ ++	Bewegt den Zeiger auf die nächste Speicherzelle (#1), erhöht den Wert um 7
>+++++ +++++	Bewegt den Zeiger auf die zweite Speicherzelle (#2), erhöht den Wert um 10
>+++	Bewegt den Zeiger auf die dritte Speicherzelle (#3), erhöht den Wert um 3
>+	Bewegt den Zeiger auf die vierte Speicherzelle (#4), erhöht den Wert um 1
<<<<-	Bewegt den Zeiger auf die erste Speicherzelle (#0) und verringert den Wert um 1
]	Ende der Schleife
>+.,	Bewegt den Zeiger auf die zweite Speicherzelle (#2) und gibt den ASCII-Code 72 ("H") aus
>+.,	Bewegt den Zeiger auf die dritte Speicherzelle (#3) und gibt den ASCII-Code 101 ("e") aus
+++++ ++.,	Erhöht den Wert der dritten Speicherzelle (#3) um 5 und gibt den ASCII-Code 108 ("l") aus
.	Gibt den ASCII-Code 108 ("l") aus
+++.,	Erhöht den Wert der dritten Speicherzelle (#3) um 3 und gibt den ASCII-Code 111 ("o") aus
>+.,	Bewegt den Zeiger auf die vierte Speicherzelle (#4) und gibt den ASCII-Code 32 (" ") aus
<<+++++ ++++++ +++++.,	Bewegt den Zeiger auf die erste Speicherzelle (#0) und gibt den ASCII-Code 87 ("W") aus

Brainfuck-Code	Aktion
>.	Bewegt den Zeiger auf die zweite Speicherzelle (#2) und gibt den ASCII-Code 111 ("o") aus
+++.	Erhöht den Wert der zweiten Speicherzelle (#2) um 3 und gibt den ASCII-Code 114 ("r") aus
-----.	Verringert den Wert der zweiten Speicherzelle (#2) um 6 und gibt den ASCII-Code 108 ("l") aus
----- --.	Verringert den Wert der zweiten Speicherzelle (#2) um 3 und gibt den ASCII-Code 100 ("d") aus
>+.	Bewegt den Zeiger auf die dritte Speicherzelle (#3) und gibt den ASCII-Code 33 ("!") aus
>.	Gibt den ASCII-Code 10 (newline) aus

Implementation mit Java

Es gibt einige wichtige Code-Ausschnitte, die wie folgt beschrieben werden können:

Einlesen des Brainfuck-Codes aus einer Datei

In diesem Abschnitt wird der Dateipfad zur Textdatei festgelegt, die den Brainfuck-Code enthält. Mit der Methode `Files.readString` wird der Inhalt der Datei in einen String gespeichert.

```
public class Main {
    public static void main(String[] args) throws IOException {
        // FileToRead
        Path FileToRead = Path.of("D:\\Studium\\IntellijProjekte\\BrainfuckReadfile\\src\\main\\resources\\mandelbrot.bf");

        // Read the file to a string
        String str = Files.readString(FileToRead);

        // Create an arraylist to store enums from the string
        ArrayList<OpcodeEnum> EnumList = new ArrayList<>();

        // Create a stack for brackets
        Stack<Integer> bracketStack = new Stack<>();
```

Speichern der Opcodes

In diesem Abschnitt wird eine Liste erstellt, die die Opcodes des Brainfuck-Codes enthält. Dabei wird jede Zeichenfolge in der Brainfuck-Syntax in eine der acht Opcodes übersetzt. Diese werden in der `ArrayList` mit

dem Namen `EnumList` gespeichert.

```
// Iterate over the string
for (int i = 0; i < str.length(); i++) {
    // Get the current character
    char c = str.charAt(i);

    // Check if the character is a known opcode and add the corresponding enum to the list
    switch (c) {
        case '>' -> {
            EnumList.add(OpcodesEnum.GRÖSSERALS);
        }
        case '<' -> {
            EnumList.add(OpcodesEnum.KLEINERALS);
        }
        case '+' -> {
            EnumList.add(OpcodesEnum.PLUS);
        }
        case '-' -> {
            EnumList.add(OpcodesEnum.MINUS);
        }
        case '.' -> {
            EnumList.add(OpcodesEnum.PUNKT);
        }
        case ',' -> {
            EnumList.add(OpcodesEnum.KOMMA);
        }
        case '[' -> {
            EnumList.add(OpcodesEnum.KLAMMERAUF);
            bracketStack.push(EnumList.size() - 1);
            bracketList.add(new BracketPair(EnumList.size() - 1, -1));
        }
        case ']' -> {
            EnumList.add(OpcodesEnum.KLAMMERZU);
            if (bracketStack.isEmpty()) {
                // Prints the position of the position of the closing bracket missing an opening bracket
                System.out.println("Keis corresponding zum closing bracket ade Position: " + i);
            } else {
                int beginIndex = bracketStack.pop();
                for (BracketPair bp : bracketList) {
                    if (bp.getBegin() == beginIndex) {
                        bp.setEnd(EnumList.size() - 1);
                        break;
                    }
                }
            }
        }
    }
}
}
```

Behandlung von Schleifen

Dieser Abschnitt behandelt die Schleifen in Brainfuck. Hierfür wird eine `Stack`-Datenstruktur verwendet, um die Positionen der öffnenden Klammern in der `ArrayList EnumList` zu speichern. Wenn eine schliessende Klammer gefunden wird, wird die Position des passenden öffnenden Klammerns aus dem Stack abgerufen

und ein neues `BracketPair`-Objekt erstellt, das die beiden Positionen speichert.

```
case '[' -> {
    EnumList.add(Opcodenum.KLAMMERAUF);
    bracketStack.push(EnumList.size() - 1);
    bracketList.add(new BracketPair(EnumList.size() - 1, -1));
}
case ']' -> {
    EnumList.add(Opcodenum.KLAMMERZU);
    if (bracketStack.isEmpty()) {
        // Prints the position of the closing bracket missing an opening bracket
        System.out.println("Keis corresponding zum closing bracket ade Position: " + i);
    } else {
        int beginIndex = bracketStack.pop();
        for (BracketPair bp : bracketList) {
            if (bp.getBegin() == beginIndex) {
                bp.setEnd(EnumList.size() - 1);
                break;
            }
        }
    }
}
```

Ausführen des Brainfuck-Codes

Dieser Abschnitt führt den Brainfuck-Code aus, indem er jedes Opcode-Element in der `ArrayList EnumList` abrufen und die entsprechenden Anweisungen ausführt. Die Anweisungen sind das Inkrementieren oder Dekrementieren des Zeigers, das Erhöhen oder Verringern des Wertes im Speicher, das Drucken oder Einlesen

von Werten und die Verarbeitung von Schleifen.

```
// Iterate over the string
for (int i = 0; i < str.length(); i++) {
    // Get the current character
    char c = str.charAt(i);

    // Check if the character is a known opcode and add the corresponding enum to the list
    switch (c) {
        case '>' -> {
            EnumList.add(OpcodesEnum.GRÖSSERALS);
        }
        case '<' -> {
            EnumList.add(OpcodesEnum.KLEINERALS);
        }
        case '+' -> {
            EnumList.add(OpcodesEnum.PLUS);
        }
        case '-' -> {
            EnumList.add(OpcodesEnum.MINUS);
        }
        case '.' -> {
            EnumList.add(OpcodesEnum.PUNKT);
        }
        case ',' -> {
            EnumList.add(OpcodesEnum.KOMMA);
        }
        case '[' -> {
            EnumList.add(OpcodesEnum.KLAMMERAUF);
            bracketStack.push(EnumList.size() - 1);
            bracketList.add(new BracketPair(EnumList.size() - 1, -1));
        }
        case ']' -> {
            EnumList.add(OpcodesEnum.KLAMMERZU);
            if (bracketStack.isEmpty()) {
                // Prints the position of the position of the closing bracket missing an opening bracket
                System.out.println("Keis corresponding zum closing bracket ade Position: " + i);
            } else {
```

Behandlung von Fehlern

Dieser Abschnitt behandelt Fehler, die während der Übersetzung oder Ausführung des Brainfuck-Codes auftreten können. Wenn beispielsweise eine schließende Klammer ohne eine entsprechende öffnende Klammer gefunden wird, wird die Position des Fehlers ausgegeben. Ebenso wird eine Warnung ausgegeben, wenn eine öffnende Klammer gefunden wird, für die es keine passende schließende Klammer gibt.

```
case ']' -> {
    EnumList.add(OpcodesEnum.KLAMMERZU);
    if (bracketStack.isEmpty()) {
        // Prints the position of the position of the closing bracket missing an opening bracket
        System.out.println("Keis corresponding zum closing bracket ade Position: " + i);
    } else {
        int beginIndex = bracketStack.pop();
        for (BracketPair bp : bracketList) {
            if (bp.getBegin() == beginIndex) {
                bp.setEnd(EnumList.size() - 1);
                break;
            }
        }
```

Klasse `BracketPair`

Dies ist eine einfache Klasse, die ein Paar von Klammern darstellt. Sie speichert die Positionen der öffnenden

```
public static class BracketPair {
    private int begin;
    private int end;

    public BracketPair(int begin, int end) {
        this.begin = begin;
        this.end = end;
    }
    public int getBegin() {
        return begin;
    }
    public int getEnd() {
        return end;
    }
    public void setEnd(int end) {
        this.end = end;
    }
}
```

und schliessenden Klammern in der `ArrayList EnumList`. }

Enumeration `OpcodeEnum`

Dies ist eine Enumeration, die die acht Opcodes des Brainfuck-Codes definiert. Sie wird verwendet, um die

```
public enum OpcodeEnum {
    GRÖSSERALS,
    KLEINERALS,
    PLUS,
    MINUS,
    PUNKT,
    KOMMA,
    KLAMMERAUF,
    KLAMMERZU
}
```

Opcodes in der `ArrayList EnumList` zu speichern. }

Implementation

Aufbau des Projekts

Das Projekt ist in 3 Komponenten gegliedert ein mal das Viewer, BF und den BrainfuckCompiler.

Viewer

Das Viewer-Projekt ist eine Software, die speziell dafür entwickelt wurde, um Text auf dem OLED-Display darzustellen. Es ist ein beliebtes Display-Format für kleine elektronische Geräte wie Smartwatches, Fitness-Tracker und vieles mehr.

Die Texte, die auf dem OLED-Display dargestellt werden sollen, werden von den beiden anderen Projekten generiert: BF und BrainfuckCompiler.

Das Viewer-Projekt fungiert als Vermittler zwischen den beiden anderen Projekten und dem OLED-Display. Sobald die Texte von BF oder BrainfuckCompiler generiert wurden, können die ans Viewer-Projekt gesendet werden. Das Viewer-Projekt nimmt diese Texte entgegen, und zeigt diese dann auf dem OLED-Display an.

Der Viewer verwendet dabei eine spezielle Programmierschnittstelle (API), um mit dem OLED-Display zu kommunizieren. Diese API ermöglicht es dem Viewer, den Text auf dem Display anzuzeigen, indem er die richtigen Befehle an das Display sendet.

Insgesamt ist das Viewer-Projekt eine wichtige Komponente in diesem System, da es die Texte von BF und BrainfuckCompiler auf dem OLED-Display darstellt und so den Entwicklern eine einfache Möglichkeit bietet, ihre Algorithmen zu testen und zu debuggen.

```

File Edit Tabs Help
Main$opcodeenum.class
Main.class
ch/

Nothing added to commit but untracked files present (use "git add" to track)
sugu@sugu:~/Documents/project-02/BrainfuckCompilerJava/src/main/java/ch/teko/wema $ cd
sugu@sugu:~$ ls
2023-03-11-154654_1920x1080_screenshot.png  2023-03-11-181749_1920x1080_screenshot.png  2023-03-11-181751_1920x1080_screenshot.png  brickd_linux_latest_armhf.deb  Downloads  Templates
2023-03-11-181734_1920x1080_screenshot.png  2023-03-11-181751_1920x1080_screenshot_000.png  2023-03-11-181752_1920x1080_screenshot_000.png  brickv_linux_latest.deb        Music       Videos
2023-03-11-181740_1920x1080_screenshot.png  2023-03-11-181751_1920x1080_screenshot_001.png  2023-03-11-181752_1920x1080_screenshot.png  Desktop                        Pictures
2023-03-11-181747_1920x1080_screenshot.png  2023-03-11-181751_1920x1080_screenshot_002.png  Bookshelf                                     Documents  Public

sugu@sugu:~$ cd Do
Documents/ Downloads/
sugu@sugu:~$ cd Do
Documents/ Downloads/
sugu@sugu:~$ cd Documents/project-02/
sugu@sugu:~/Documents/project-02$ cd viewer/
sugu@sugu:~/Documents/project-02/viewer$ ls
Cargo.lock  Cargo.toml  data  run.sh  src  target
sugu@sugu:~/Documents/project-02/viewer$ cat run.sh
#!/usr/bin/env sh
cargo run --release -- \
  --host localhost \
  --port 4223 \
  --uid Hit \
  --path ./data/mandelbrot.txt
sugu@sugu:~/Documents/project-02/viewer$ ls data/
feer  bfeer.txt  dbff2c  dbff2c.txt  HelloWorld.txt  mandelbrot.txt  squares  squares.txt
sugu@sugu:~/Documents/project-02/viewer$ nvim run.sh
sugu@sugu:~/Documents/project-02/viewer$ cat run.sh ^C
sugu@sugu:~/Documents/project-02/viewer$ ./run.sh
Finished release [optimized] tar 1 (s) in 0.94s
Running target/release/viewer --host localhost --port 4223 --uid Hit --path ./data/HelloWorld.txt
Hello, World!
sugu@sugu:~/Documents/project-02/viewer$

```

Dies ist der Output der Textdatei. Diese Information dient als Debug-Funktion, sodass wir feststellen können, ob es auf dem Display korrekt angezeigt wird.

bf

Das Projekt nennt sich "Brainfuck Transpiler". Es ist eine Anwendung, die geschrieben wurde, um Brainfuck-Code in C- und Rust-Code umzuwandeln. Der Zweck dieser Anwendung ist es, Brainfuck-Code in eine höhere Programmiersprache zu übersetzen, die einfacher zu lesen und zu verstehen ist.

Die Anwendung besteht aus fünf Hauptkomponenten:

Tokenizer:

Der Tokenizer ist für das Aufteilen des Brainfuck-Codes in Tokens zuständig. Ein Token ist eine Art von Textelement, das eine spezielle Bedeutung hat. Der Tokenizer durchsucht den Brainfuck-Code und identifiziert jedes Token. Jedes Token wird dann an den Parser weitergegeben.

Parser:

Der Parser nimmt die Tokens welche er verwendet, um IR-Expressions zu generieren. Die generierten IR-Expressions werden dann an den Optimizer weitergegeben, der sie optimiert.

Optimizer:

Der Optimizer ist ein Programmteil/Funktion, die Brainfuck-Code analysiert und optimiert, um die Ausführung des Programms zu verbessern, also zu optimieren. Der Zweck des Optimierers besteht darin, die Ausführungszeit und den Speicherverbrauch des Programms zu reduzieren, indem der Code effizienter gestaltet wird.

Pipeline:

Die Pipeline nimmt den Brainfuck-Code als Eingabe und gibt einen optimierten Brainfuck-Code als Ausgabe aus. Der optimierte Code kann dann in eine ausführbare Datei kompiliert werden. In unserem Fall wird dies von den Backends VM, Rust und C übernommen.

Transpiler:

Der Transpiler ist das Kernstück der Anwendung. Der Transpiler ruft sowohl das gewünschte Backend auf. Der Transpiler liest die Expressions ein und sorgt dafür dass der Korrekte code erzeugt wird für das jeweilige backend. Der Transpiler übersetzt die Tokens in C- oder Rust-Code, der dann in einen String geschrieben wird.

Das Ergebnis des Projekts ist eine C- oder Rust-Datei, die den übersetzten Brainfuck-Code enthält. Diese Datei kann dann in eine ausführbare Datei kompiliert werden und auf einer beliebigen Plattform ausgeführt werden.

Dieses Projekt kann Entwicklern helfen, Brainfuck-Code auf eine höhere Programmiersprache zu übersetzen und so das Debugging und die Wartung von Brainfuck-Code zu vereinfachen.

Tokenizer

Der Tokenizer analysiert die vordefinierten Zeichen von Brainfuck und erzeugt entsprechende Tokens für jeden von ihnen.

```
pub struct Tokenizer;

impl Tokenizer {
    pub fn tokenize(text: &str) -> Vec<Token> {
        text.chars().into_iter().map(Self::tokenize_char).collect()
    }

    fn tokenize_char(char: char) -> Token {
        match char {
            '+' => Token::Plus,
            '-' => Token::Minus,
            '.' => Token::Dot,
            '>' => Token::Shr,
            '<' => Token::Shl,
            '[' => Token::OpenBracket,
            ']' => Token::CloseBracket,
            _ => Token::Whitespace(char),
        }
    }
}
```

Parser

Für jedes Token erstellt der Parser eine Expression.

```
pub struct Parser;

impl Parser {
    pub fn parse(tokens: &[Token]) -> Vec<Expression> {
        let mut expressions = vec![];
        let mut indexes: Vec<usize> = vec![];

        for token in tokens.iter().filter(Self::filter) {
            match token {
                Token::Plus => {
                    expressions.push(Expression::IncVal(1));
                }
                Token::Minus => {
                    expressions.push(Expression::DecVal(1));
                }
                Token::Dot => {
                    expressions.push(Expression::Output);
                }
                Token::Comma => {
                    expressions.push(Expression::Input);
                }
                Token::Shr => {
                    expressions.push(Expression::IncPtr(1));
                }
                Token::Shl => {
                    expressions.push(Expression::DecPtr(1));
                }
                Token::OpenBracket => {
                    indexes.push(expressions.len());
                }
                Token::CloseBracket => {
                    let start_index = indexes.pop().unwrap();
                    let r#loop = expressions.split_off(start_index);
                    expressions.push(Expression::Loop(r#loop));
                }
                Token::Whitespace(_) => {
                    unreachable!()
                }
            }
        }
        expressions
    }

    fn filter(token: &&Token) -> bool {
        !matches!(token, Token::Whitespace(_))
    }
}
```

Pipeline

Backends

Die Backends sind dafür verantwortlich, auf Basis der Expressions das entsprechende Backend zu generieren. In unserem Fall handelt es sich um Rust- und C-Dateien, die aus den Expressions erzeugt werden. Die Backends übernehmen dabei die Aufgabe, den erzeugten Code zu transpilieren. Sobald das Backend erstellt wurde, kann der transpilierte/generierte Code kompiliert und ausgeführt werden. In Rust wird dies gemacht mit `rustc` und in C mit `gcc/clang`.

IR

Die Intermediäre Repräsentation (IR) stellt den Quellcode in einer vereinfachten und standardisierten Form dar, die unabhängig von einer bestimmten Programmiersprache ist. Sie dient dazu, den Quellcode auf seine Bedeutung hin zu analysieren und zu optimieren, bevor er in Maschinencode umgewandelt wird.

Die Verwendung von IR erleichtert es auch, verschiedene Optimierungstechniken auf den Code anzuwenden, ohne dass dabei die eigentliche Programmiersprache berücksichtigt werden muss. Auf diese Weise kann der Compiler oder das Programmierwerkzeug eine optimierte Version des Codes generieren, die schneller und effizienter ausgeführt werden kann.

IR Code

```
pub enum Expression {  
    IncVal(u8),  
    DecVal(u8),  
    IncPtr(usize),  
    DecPtr(usize),  
    MulVal(usize, u8),  
    Clear,  
    Loop(Vec<Expression>),  
    Output,  
    Input,  
}
```

IR Erklärung

Der Code definiert ein öffentliches Enum namens "Expression". Enums sind eine Konstruktion, welche ermöglichen, eine begrenzte Anzahl von möglichen Werten zu definieren. Diese stellen dann ein bestimmtes Konzept dar.

In diesem Fall enthält die "Expression" Enum sieben Varianten:

"IncVal" und "DecVal" haben beide einen Parameter vom Typ "u8". Diese Varianten repräsentieren die Operationen, um den Wert an der aktuellen Speicherposition im Speicher um 1 zu erhöhen oder zu verringern.

"IncPtr" und "DecPtr" haben beide einen Parameter vom Typ "usize". Diese Varianten repräsentieren die Operationen, um den Speicherzeiger um 1 nach rechts oder links zu verschieben.

"MulVal" hat einen Parameter vom Typ "isize" und einen Parameter vom Typ "u8". Diese Variante repräsentiert die Operation, bei der der Wert an der aktuellen Speicherposition im Speicher mit einem gegebenen Wert multipliziert wird.

"Clear" hat keine Parameter und repräsentiert die Operation, um den Wert an der aktuellen Speicherposition im Speicher auf 0 zu setzen.

"Loop" hat einen Parameter vom Typ "Vec". Diese Variante repräsentiert eine Schleife, die aus einer Liste von Ausdrücken besteht, die wiederholt ausgeführt werden, solange der Wert an der aktuellen Speicherposition im Speicher ungleich 0 ist.

"Output" und "Input" haben keine Parameter. Diese Varianten repräsentieren die Operationen, um den Wert an der aktuellen Speicherposition im Speicher als ASCII-Zeichen auf der Konsole auszugeben oder ein ASCII-Zeichen von der Konsole einzulesen und an der aktuellen Speicherposition im Speicher zu speichern.

Zusammengefasst definiert dieser Code also enums, die verschiedene Operationen für die Manipulation des Speichers in der Programmiersprache Brainfuck für die Ausführung auf einer Maschine definiert.

Optimierungen

Der Optimizer besteht aus drei Optimierungen: Copy, Clear und ConcatOptimizer.

ConcatOptimizer

```
struct ConcatOptimizer;

impl ConcatOptimizer {
    fn optimize_stage_01(expressions: &[Expression]) -> Vec<Expression> {
        let mut optimized = vec![];
        for expression in expressions {
            match (expression, optimized.last()) {
                (Expression::IncVal(1), Some(&Expression::IncVal(amount))) => {
                    replace_last(&mut optimized, Expression::IncVal(amount + 1))
                }
                (Expression::DecVal(1), Some(&Expression::DecVal(amount))) => {
                    replace_last(&mut optimized, Expression::DecVal(amount + 1))
                }
                (Expression::IncPtr(1), Some(&Expression::IncPtr(amount))) => {
                    replace_last(&mut optimized, Expression::IncPtr(amount + 1))
                }
                (Expression::DecPtr(1), Some(&Expression::DecPtr(amount))) => {
                    replace_last(&mut optimized, Expression::DecPtr(amount + 1))
                }
                (Expression::Loop(expressions), _) => {
                    optimized.push(Expression::Loop(Self::optimize_stage_01(expressions)))
                }
                (expression, _) => optimized.push(expression.clone()),
            }
        }
        optimized
    }
}
```

```

    }

    fn optimize_stage_02(expressions: &[Expression]) -> Vec<Expression> {
        let mut optimized = vec![];
        for expression in expressions {
            match (expression, optimized.last()) {
                (Expression::IncVal(val), Some(&Expression::IncVal(amount))) => {
                    replace_last(&mut optimized, Expression::IncVal(amount + val))
                }
                (Expression::IncVal(val), Some(&Expression::DecVal(amount))) => {
                    concat_match!(optimized, val, DecVal, &amount, IncVal);
                }
                (Expression::DecVal(val), Some(&Expression::DecVal(amount))) => {
                    replace_last(&mut optimized, Expression::DecVal(amount + val))
                }
                (Expression::DecVal(val), Some(&Expression::IncVal(amount))) => {
                    concat_match!(optimized, val, IncVal, &amount, DecVal);
                }
                (Expression::IncPtr(val), Some(&Expression::IncPtr(amount))) => {
                    replace_last(&mut optimized, Expression::IncPtr(amount + val))
                }
                (Expression::IncPtr(val), Some(&Expression::DecPtr(amount))) => {
                    concat_match!(optimized, val, DecPtr, &amount, IncPtr);
                }
                (Expression::DecPtr(val), Some(&Expression::DecPtr(amount))) => {
                    replace_last(&mut optimized, Expression::DecPtr(amount + val))
                }
                (Expression::DecPtr(val), Some(&Expression::IncPtr(amount))) => {
                    concat_match!(optimized, val, IncPtr, &amount, DecPtr);
                }
                (Expression::Loop(expressions), _) => {
                    let sub_expressions = Self::optimize_stage_02(expressions);
                    if !sub_expressions.is_empty() {
                        optimized.push(Expression::Loop(Self::optimize_stage_02(&sub_expressions)))
                    }
                }
            }
            (expression, _) => optimized.push(expression.clone()),
        }
        optimized
    }
}

impl Optimizer for ConcatOptimizer {
    fn optimize(expressions: &[Expression]) -> Vec<Expression> {
        let expressions = ConcatOptimizer::optimize_stage_01(expressions);

        ConcatOptimizer::optimize_stage_02(&expressions)
    }
}

```

ConcatOptimizer Erklärung

Der ConcatOptimizer enthält zwei Funktionen: `optimize_stage_01` und `optimize_stage_02`, die jeweils ein Vektor von Ausdrücken (Expression) als Eingabe nehmen und einen optimierte Vektor von Ausdrücken zurückgeben.

`optimize_stage_01` optimiert die Ausdrücke in Bezug auf die Inkrementierung/Dekrementierung von Werten und Zeigern sowie auf Schleifen. Die Optimierung wird durchgeführt, indem ähnliche aufeinanderfolgende Ausdrücke zusammengefasst werden. Beispiel: Wenn der Ausdruck `IncVal(1)` direkt auf einen `IncVal(n)` Ausdruck folgt, wird der `IncVal(n)` Ausdruck durch einen einzigen `IncVal(n+1)` Ausdruck ersetzt.

`optimize_stage_02` führt eine weitere Optimierung der Ausdrücke durch, indem sie ähnliche aufeinanderfolgende Ausdrücke miteinander kombiniert. Beispiel: Wenn der Ausdruck `IncVal(n)` direkt auf einen `DecVal(m)` Ausdruck folgt, werden beide Ausdrücke durch einen einzigen `Loop(IncVal(n-m))` Ausdruck ersetzt.

Schliesslich implementiert die Struktur die `optimize` Funktion, die eine Kette von Optimierungen ausführt, indem sie zuerst `optimize_stage_01` auf die Eingabe anwendet und dann das Ergebnis an `optimize_stage_02` weitergibt.

ClearOptimizer

```
struct ClearOptimizer;

impl Optimizer for ClearOptimizer {
    fn optimize(expressions: &[Expression]) -> Vec<Expression> {
        let mut optimized: Vec<Expression> = vec![];

        for expression in expressions {
            match expression {
                Expression::Loop(expressions) => match expressions[..] {
                    [Expression::DecVal(1)] | [Expression::IncVal(1)] => {
                        optimized.push(Expression::Clear)
                    }
                    _ => {
                        let mut sub_optimized = vec![];
                        let sub_expressions =
                            ClearOptimizer::optimize(expressions);
                        sub_optimized.extend(sub_expressions);

                        if !sub_optimized.is_empty() {
                            optimized.push(Expression::Loop(sub_optimized));
                        }
                    }
                },
                _ => {
                    optimized.push(expression.clone());
                }
            }
        }
        optimized
    }
}
```

```

    }
}

```

Die optimize-Methode nimmt eine Liste von Expression-Werten und gibt eine optimierte Liste von Expression-Werten zurück. Der Optimierer entfernt alle Schleifen, die lediglich aus einer einzigen Inkrementierung oder Dekrementierung des Zellwerts bestehen, und ersetzt sie durch einen einzigen Clear-Befehl.

Die Funktion iteriert durch die gegebene Liste von Expressions und prüft, ob die Expression eine Schleife ist. Ist dies der Fall: es wird überprüft, ob die Schleife aus einer einzigen Inkrementierung oder Dekrementierung besteht. In diesem Fall wird ein Clear-Befehl zur optimierten Liste hinzugefügt. Andernfalls wird die Schleife rekursiv optimiert und zur optimierten Liste hinzugefügt.

Wenn die Expression keine Schleife ist, wird sie unverändert zur optimierten Liste hinzugefügt.

Am Ende gibt die Funktion die optimierte Liste von Expression-Werten zurück.

CopyOptimizer

```

struct CopyOptimizer;

impl Optimizer for CopyOptimizer {
    fn optimize(expressions: &[Expression]) -> Vec<Expression> {
        let mut optimized = vec![];

        for expression in expressions {
            match expression {
                Expression::Loop(r#loop) => {
                    let mut loop_optimized = vec![];
                    let mut context =
                        CopyOptimizerContext::new(optimized.len().wrapping_sub(1)
% usize::MAX);

                    for expression in r#loop {
                        match expression {
                            Expression::Clear => {
                                loop_optimized.push(expression.clone());
                                context.set_side_effect(true);
                            }
                            Expression::IncVal(val) => {
                                loop_optimized.push(expression.clone());
                                context.add_inc_val(*val);
                            }
                            Expression::DecVal(val) => {
                                loop_optimized.push(expression.clone());
                                context.add_dec_val(*val);
                            }
                            Expression::MulVal(_, _) => {
                                loop_optimized.push(expression.clone());
                                context.set_side_effect(true);
                            }
                            Expression::IncPtr(val) => {
                                loop_optimized.push(expression.clone());

```

```

        context.add_inc_ptrs(*val);
    }
    Expression::DecPtr(val) => {
        loop_optimized.push(expression.clone());
        context.add_dec_ptrs(*val);
    }
    Expression::Loop(r#loop) => {
        loop_optimized
            .extend(Self::optimize(&
[Expression::Loop(r#loop.clone())]));
        context.set_side_effect(true);
    }
    Expression::Output => {
        loop_optimized.push(expression.clone());
        context.set_side_effect(true);
    }
    Expression::Input => {
        loop_optimized.push(expression.clone());
        context.set_side_effect(true);
    }
    }
}

if let Some(expressions) = context.generate_expressions() {
    optimized.extend(expressions);
} else {
    optimized.push(Expression::Loop(loop_optimized))
}
_ => {
    optimized.push(expression.clone());
}
}
}

optimized
}
}

```

Die CopyOptimizer-Struktur implementiert die Optimizer-Trait. Die Methode optimize erhält einen Slice von Expression-Instanzen und gibt eine optimierte Version zurück.

Die Optimierung, versucht die aufeinanderfolgenden Operationen, die eine Kopie von Daten ausführen, zusammenzufassen. Die optimierten Ausdrücke werden in einem Vektor namens "optimized" gesammelt.

In der Methode optimize wird für jede Expression in der übergebenen Slice einen Match-Block durchlaufen, um den Expression-Typ zu bestimmen und die Optimierung durchzuführen. Wenn der Expression-Typ eine Schleife ist, wird für jede Expression in der Schleife einen weiteren Match-Block durchlaufen, um zu bestimmen, welche Art von Expression vorliegt.

Wenn eine Expression vorliegt, die eine Kopie von Daten ausführt, wird ein CopyOptimizerContext erstellt, um Informationen über die Ausführung dieser Expression zu sammeln.

Am Ende der Schleife wird geprüft, ob es sich lohnt, die Operationen zusammenzufassen. Wenn ja, wird eine optimierte Version der aufeinanderfolgenden Operationen erzeugt. Andernfalls wird die Schleife unverändert in die optimierte Version eingefügt.

Der Code ist relativ komplex und erfordert ein gutes Verständnis der Brainfuck-Syntax und der Optimierungstechniken, um vollständig verstanden zu werden.

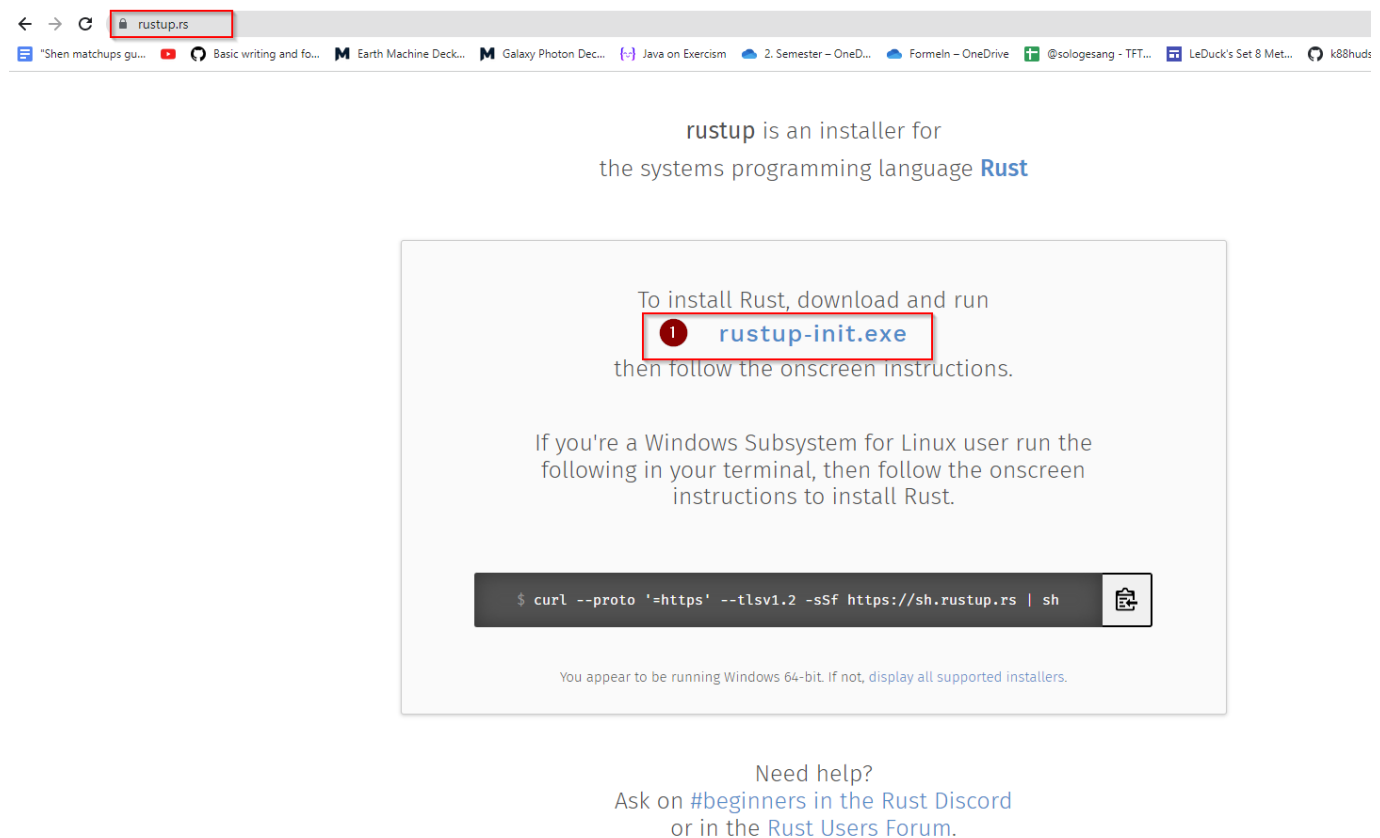
Tests

Um die Implementationen zu Testen, werden wir unsere Ausgaben mit den Ausgaben des Online Brainfuck compilers vergleichen: <https://copy.sh/brainfuck/>

Rust Installation

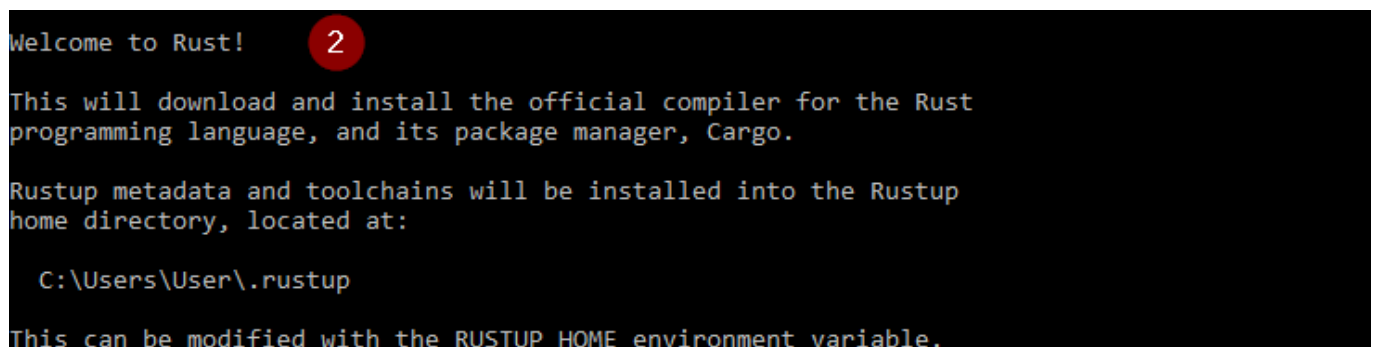
Um Rust auf einem Windows computer auszuführen, müssen wir zuerst einen Rust-compiler installieren. Dafür navigieren wir auf rustup.rs.

Dann laden wir die Datei herunter für unser Betriebssystem.



The screenshot shows a web browser window with the address bar displaying 'rustup.rs'. The main content of the page states: 'rustup is an installer for the systems programming language Rust'. Below this, a box contains instructions: 'To install Rust, download and run rustup-init.exe then follow the onscreen instructions.' A red circle with the number '1' highlights 'rustup-init.exe'. Further down, it says: 'If you're a Windows Subsystem for Linux user run the following in your terminal, then follow the onscreen instructions to install Rust.' A terminal snippet shows the command: '\$ curl --proto 'https' --tlsv1.2 -sSf https://sh.rustup.rs | sh'. At the bottom, it says: 'You appear to be running Windows 64-bit. If not, display all supported installers.' Below the box, there is a link for help: 'Need help? Ask on #beginners in the Rust Discord or in the Rust Users Forum.'

Als nächstes bestätigen wir die Installation von Rust.



The screenshot shows a terminal window with the following text: 'Welcome to Rust!' followed by a red circle with the number '2'. The text continues: 'This will download and install the official compiler for the Rust programming language, and its package manager, Cargo. Rustup metadata and toolchains will be installed into the Rustup home directory, located at: C:\Users\User\.rustup This can be modified with the RUSTUP_HOME environment variable.'

The Cargo home directory is located at:

```
C:\Users\User\.cargo
```

This can be modified with the CARGO_HOME environment variable.

The cargo, rustc, rustup and other commands will be added to Cargo's bin directory, located at:

```
C:\Users\User\.cargo\bin
```

This path will then be added to your PATH environment variable by modifying the HKEY_CURRENT_USER/Environment/PATH registry key.

You can uninstall at any time with `rustup self uninstall` and these changes will be reverted.

Current installation options:

```
default host triple: x86_64-pc-windows-msvc
default toolchain: stable (default)
profile: default
modify PATH variable: yes
```

- 1) Proceed with installation (default)
- 2) Customize installation
- 3) Cancel installation

>1

```
info: profile set to 'default'
info: default host triple is x86_64-pc-windows-msvc
info: syncing channel updates for 'stable-x86_64-pc-windows-msvc'
info: latest update on 2023-03-09, rust version 1.68.0 (2c8cc3432 2023-03-06)
info: downloading component 'cargo'
info: downloading component 'clippy'
info: downloading component 'rust-docs'
info: downloading component 'rust-std'
info: downloading component 'rustc'
 63.9 MiB / 63.9 MiB (100 %) 34.4 MiB/s in 1s ETA: 0s
info: downloading component 'rustfmt'
info: installing component 'cargo'
info: installing component 'clippy'
info: installing component 'rust-docs'
 19.4 MiB / 19.4 MiB (100 %) 3.6 MiB/s in 4s ETA: 0s
info: installing component 'rust-std'
 27.6 MiB / 27.6 MiB (100 %) 14.7 MiB/s in 1s ETA: 0s
info: installing component 'rustc'
 63.9 MiB / 63.9 MiB (100 %) 16.6 MiB/s in 3s ETA: 0s
info: installing component 'rustfmt'
info: default toolchain set to 'stable-x86_64-pc-windows-msvc'

  stable-x86_64-pc-windows-msvc installed - rustc 1.68.0 (2c8cc3432 2023-03-06)
```

Rust is installed now. Great!

To get started you may need to restart your current shell. This would reload its PATH environment variable to include Cargo's bin directory (%USERPROFILE%\\.cargo\bin).

Damit ist die Installation abgeschlossen und wir starten die shell neu. Danach können wir mit den ersten Tests beginnen.

Testcases

Wir werden 3 Files testen und die Ausgaben vom online-compiler, des Java-Compilers und des Rust-Compilers vergleichen um sicherzustellen, dass unsere Implementationen korrekt sind.

TestFile 1: Hello World

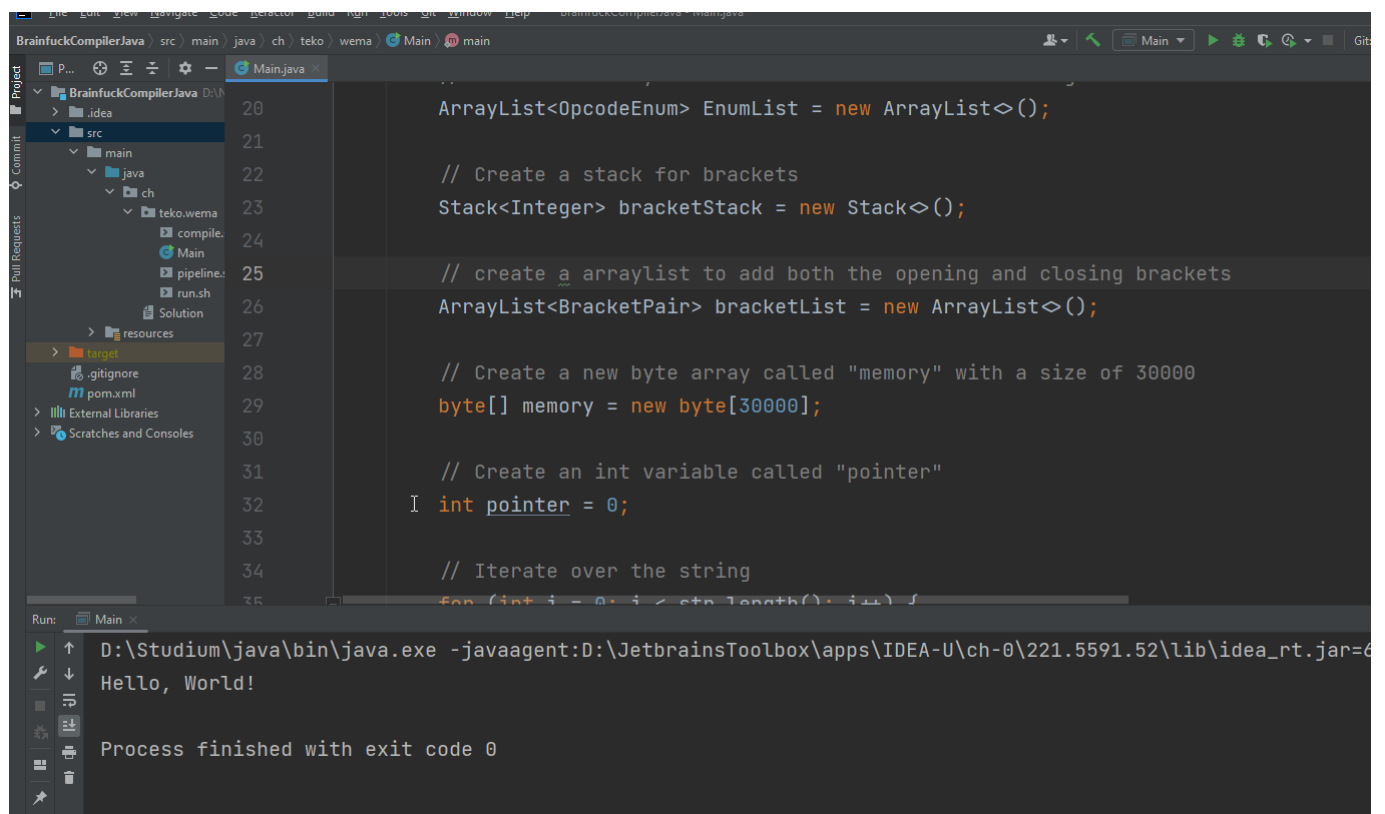
Input Online Compiler

```
1. >++++++[<++++++>-]<.>++++[<++++>-]<+.>++++.++>++++[<++++>-]<+
2. +.----->++++[<++++>-]<+.<+.----->++++[<++++>-
3. ]<+.
```

Output online Compiler

```
Hello, world!
```

Output Java



The screenshot shows an IDE window with a project named 'BrainfuckCompilerJava'. The file 'Main.java' is open, displaying the following Java code:

```
20 ArrayList<OpcodeEnum> EnumList = new ArrayList<>();
21
22 // Create a stack for brackets
23 Stack<Integer> bracketStack = new Stack<>();
24
25 // create a arraylist to add both the opening and closing brackets
26 ArrayList<BracketPair> bracketList = new ArrayList<>();
27
28 // Create a new byte array called "memory" with a size of 30000
29 byte[] memory = new byte[30000];
30
31 // Create an int variable called "pointer"
32 int pointer = 0;
33
34 // Iterate over the string
35 for (int i = 0; i < str.length(); i++) {
```

The 'Run' tab at the bottom shows the execution command and output:

```
D:\Studium\java\bin\java.exe -javaagent:D:\JetbrainsToolbox\apps\IDEA-U\ch-0\221.5591.52\lib\idea_rt.jar=6
Hello, World!
Process finished with exit code 0
```

Output Rust

The screenshot shows an IDE with a project structure on the left and a Rust source file in the center. The project structure includes a 'data' directory with a 'generated' subdirectory containing 'bfeer.exe', 'bfeer.pdb', 'bfeer.rs', 'HelloWorld.exe', 'HelloWorld.pdb', and 'HelloWorld.rs'. The 'HelloWorld.rs' file is open in the editor, showing Rust code that uses the 'bfeer' transpiler to process a program file. The code includes imports for 'std', 'fs', 'io', and 'path', and defines a 'main' function that sets environment variables, parses arguments, and executes the transpiler. The terminal at the bottom shows the command 'rustc -O -o HelloWorld.exe HelloWorld.rs' and the output 'Hello, World!'.

```

1  use std::{
2      fs::File,
3      io::{Read, Write},
4      path::{Path, PathBuf},
5  };
6
7  use bf::backends::transpilers::rust::Transpiler, core::pipeline::Pipeline;
8
9  use clap::Parser;
10
11  #[derive(Parser, Debug)]
12  #[command(author, version, about, long_about = None)]
13  struct Args {
14      #[arg(short, long)]
15      program_file: String,
16
17      #[arg(short, long)]
18      output_directory: String,
19  }
20
21  fn main() -> std::io::Result<> {
22      std::env::set_var( key: "RUST_BACKTRACE", value: "1");
23      let args :Args = Args::parse();
24
25      let mut text :String = String::new();
26      let path :&Path = Path::new( &args.program_file);
27      let mut file :File = File::open( path &args.program_file)?;
28      let _ = file.read_to_string( buf: &mut text);
29
30      let expressions :Vec<Expression> = Pipeline::execute(&text);
31      let code :String = Transpiler::transpile(&expressions);
32
33      let mut path_buf :PathBuf = PathBuf::from( &args.output_directory);
34      path_buf.push( path: path.file_name().unwrap());
35      path_buf.set_extension("rs");
36
37      let output_file_path :&Path = path_buf.as_path();
38      let mut file :File = File::create(output_file_path)?;
39      file.write_all( buf: code.as_bytes());
40
41      Ok(())
42  }

```

```

PS D:\Dev\bf\data\generated> rustc -O -o HelloWorld.exe HelloWorld.rs
warning: unused macro definition: 'loop'
--> .\HelloWorld.rs:34:14
34 | macro_rules! r#loop {
   | ^^^^^^^^^
   = note: '[warn(unused_macros)]' on by default

warning: 1 warning emitted

PS D:\Dev\bf\data\generated> .\HelloWorld.exe
Hello, World!
PS D:\Dev\bf\data\generated>

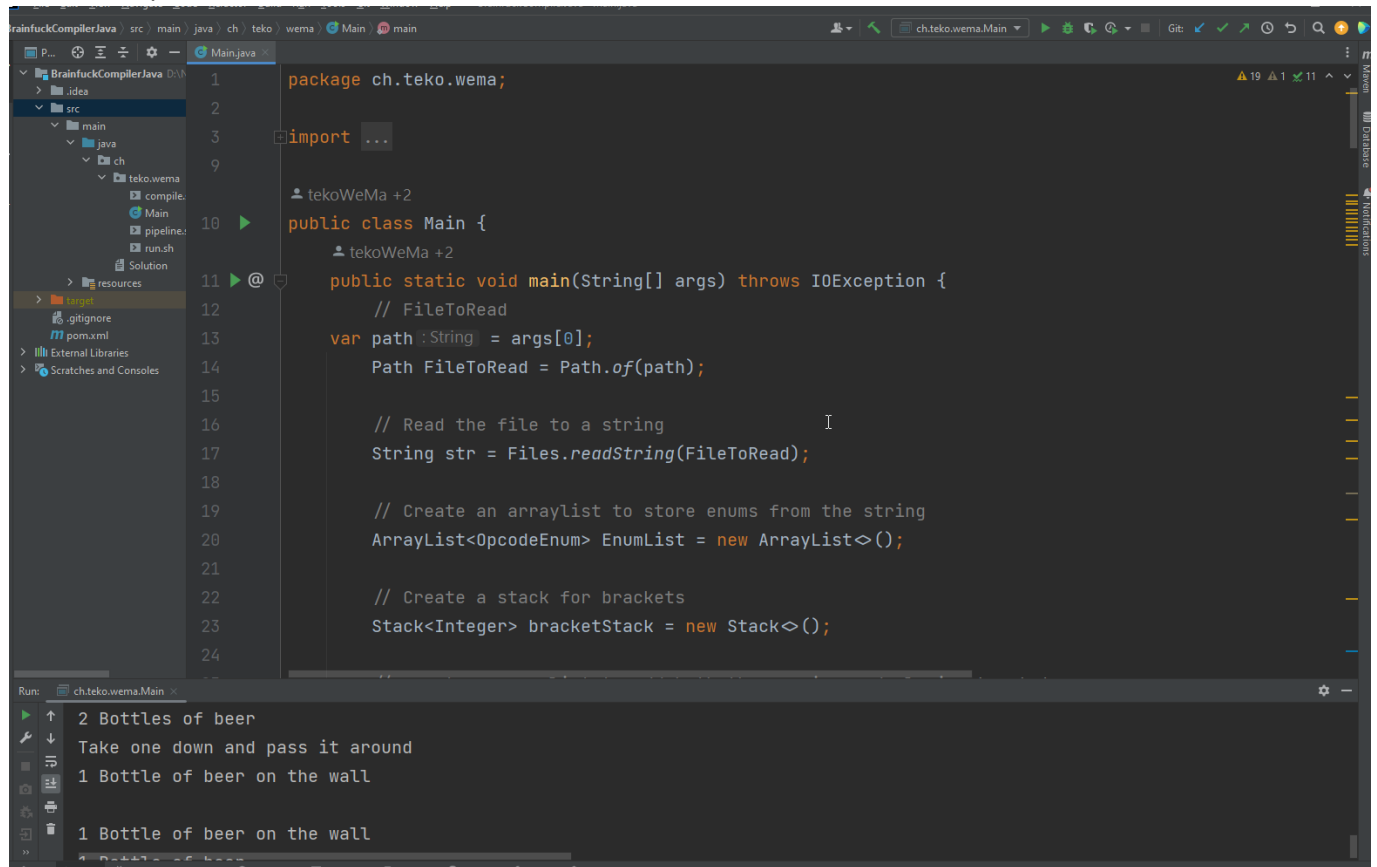
```

TestFile 2: bfeer

Input: ganzer Input in **bfeerInput**

Output Online Compiler

Ganzer Output Java in bfeer



The screenshot shows an IDE with a Java project. The package is `ch.teko.wema`. The `Main` class contains the following code:

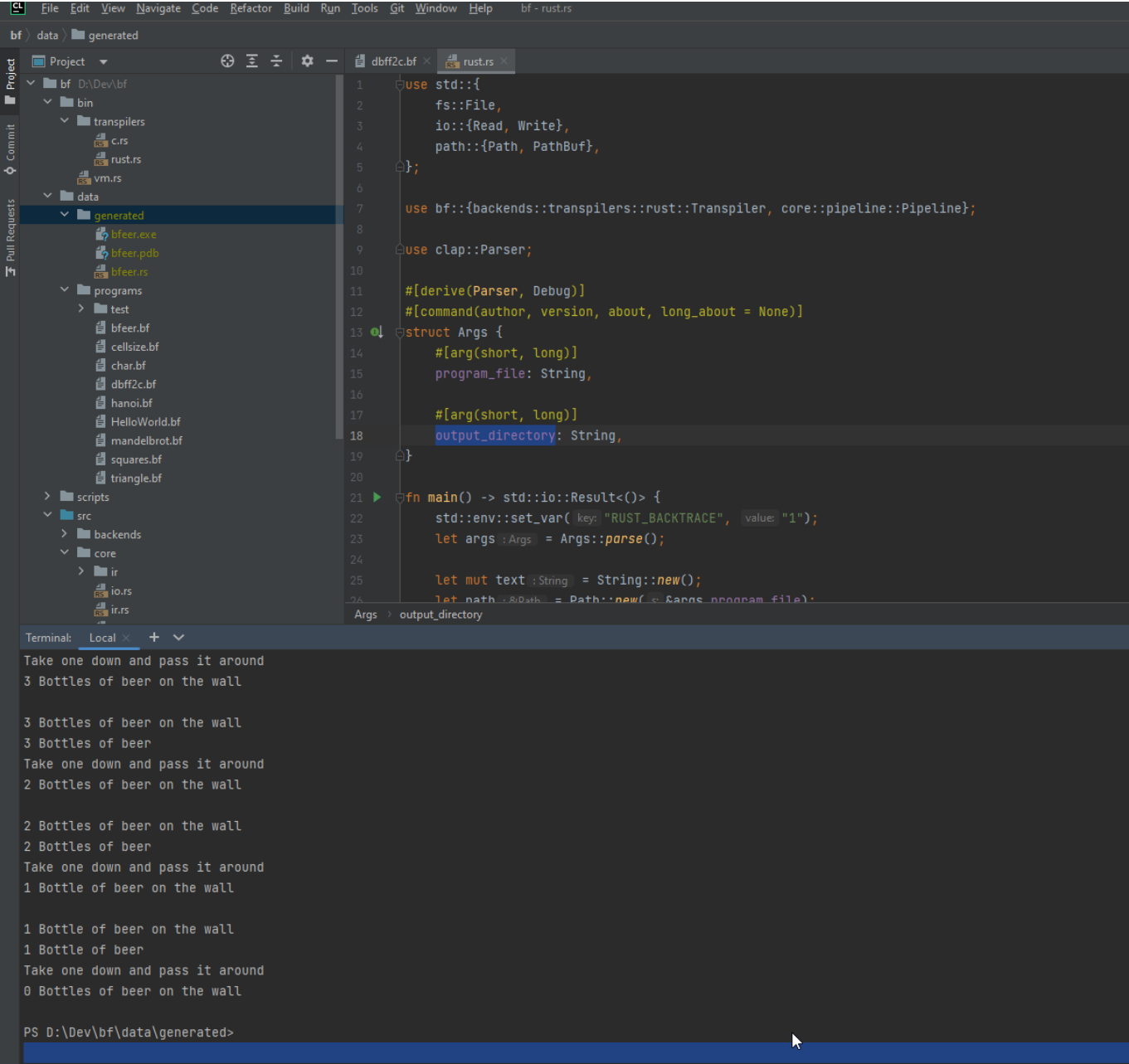
```
1 package ch.teko.wema;
2
3 import ...
4
5 tekoWeMa +2
6
7 public class Main {
8     tekoWeMa +2
9
10    public static void main(String[] args) throws IOException {
11        // FileToRead
12        var path :String = args[0];
13        Path FileToRead = Path.of(path);
14
15        // Read the file to a string
16        String str = Files.readString(FileToRead);
17
18        // Create an arraylist to store enums from the string
19        ArrayList<OpcodeEnum> EnumList = new ArrayList<>();
20
21        // Create a stack for brackets
22        Stack<Integer> bracketStack = new Stack<>();
23
24    }
```

The Run console shows the output of the program:

```
Run: ch.teko.wema.Main
2 Bottles of beer
Take one down and pass it around
1 Bottle of beer on the wall
1 Bottle of beer on the wall
```

Output Rust

Ganzer Output Rust in [bfeer](#)



TestFile 3: Mandelbrot

Output Online

[illegible]

Ganzer Output Java in Mandelbrot

[illegible]

40 / 44

Ganzer Output Rust in Mandelbrot

The screenshot shows an IDE with a project structure on the left and a terminal at the bottom. The project structure includes files like `HelloWorld.rs`, `Mandelbrot.exe`, `Mandelbrot.pdb`, and `Mandelbrot.rs`, along with various test and script files. The main code file `Mandelbrot.rs` contains Rust code for a pipeline-based transpiler. The terminal shows the execution of `rustc` and `./Mandelbrot.exe`, which results in a panic error: `thread 'main' panicked at 'called 'Option::unwrap()' on a 'None' value', ./Mandelbrot.rs:412:17`. A red box highlights this error message.

```

let expressions :Vec<Expression> = Pipeline::execute(&text);
let code :String = Transpiler::transpile(&expressions);

let mut path_buf :PathBuf = PathBuf::from( &args.output_directory);
path_buf.push( path: path.file_name().unwrap());
path_buf.set_extension("rs");

let output_file_path :&Path = path_buf.as_path();
let mut file :File = File::create(output_file_path)?;
file.write_all( buf: code.as_bytes());

Ok(())

```

```

34 | macro_rules! r#loop {
    |         ^^^^^^^
    |
    = note: `#[warn(unused_macros)]` on by default

warning: 1 warning emitted

PS D:\Dev\bf\data\generated> .\HelloWorld.exe
Hello, World!
PS D:\Dev\bf\data\generated> rustc -O .\Mandelbrot.rs
PS D:\Dev\bf\data\generated> .\Mandelbrot.exe
thread 'main' panicked at 'called 'Option::unwrap()' on a 'None' value', ./Mandelbrot.rs:412:17
note: run with 'RUST_BACKTRACE=1' environment variable to display a backtrace
PS D:\Dev\bf\data\generated>

```

Wir kriegen eine Fehlermeldung beim ersten Ausführen der bf- Datei. Dies ist der Fall, weil wir beim Optimizer nicht berücksichtigen, dass der Index negativ werden kann. Wir brauchen in diesem Fall einen höheren Offset als 0.

Dies machen wir Folgendermassen in der rs Datei:

The screenshot shows the same IDE with the `Mandelbrot.rs` file open. The code is annotated with red boxes and numbers. Box 1 highlights the `Mandelbrot.rs` file in the project structure. Box 2 highlights the `let mut pointer : usize = 100 as usize;` line in the `main` function, indicating the starting index for memory access.

```

41 | };
42 | }
43 |
44 | macro_rules! clear {
45 |     ($memory:expr, $index:expr) => {
46 |         $memory[$index] = 0;
47 |     };
48 | }
49 |
50 | macro_rules! output {
51 |     ($memory:expr, $pointer:expr) => {
52 |         print!("{}", $memory[$pointer] as char);
53 |     };
54 | }
55 |
56 | fn main() {
57 |     let mut pointer : usize = 100 as usize;
58 |     let mut memory :[u8; 30000] = [0 as u8; 30_000];
59 |
60 |     inc_val_by!(memory, pointer, 13);
61 |     mul_val_by!(memory, pointer, 1, 2);
62 |     mul_val_by!(memory, pointer, 4, 5);
63 |     mul_val_by!(memory, pointer, 5, 2);
64 |     mul_val_by!(memory, pointer, 6, 1);
65 |     clear!(memory, pointer);
66 |     inc_ptr_by!(pointer, 5);
67 |     inc_val_by!(memory, pointer, 6);
68 |     inc_ptr_by!(pointer, 1);

```

Danach bekommen wir unseren erwarteten Output:

```
PS D:\dev\bf\data\generated> .\Mandelbrot.exe
AAAAAAAAAAAAAAAAABBBBBBBBBBBBBBBBBCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCDDDDDDDDDDDEFFFFEEEDDDDDDDCCCCCCCCBBBBBBBBBBBBBBBBBBBBBBBB
AAAAAAAAAAAAAAAAABBBBBBBBBBBBBBBBBCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCDDDDDDDDDDDEEFFGIIIGFFEEEDDDDDDDCCCCCCCCBBBBBBBBBBBBBBBBBBBBBB
AAAAAAAAAAAAAAAAABBBBBBBBBBBBBBBBBCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCDDDDDDDDDDDEEEFFFI KHGGGHEGDDDDDDDDCCCCCCCCBBBBBBBBBBBBBBBBBBBBBB
AAAAAAAAAAAAAAAAABBBBBBBBBBBBBBBBBCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCDDDDDDDDDDDEEEFFFGHINTKLZOGFEEDDDDDDDCCCCCCCCBBBBBBBBBBBBBBBBBB
AAAAAAAAAAAAAAAAABBBBBBBBBBBBBBBBBCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCDDDDDDDDDDDEEEFFFGHHIKPKIHFEEEDDDDDDDCCCCCCCCBBBBBBBBBBBBBBBBBB
AAAAAAAAAAAAAAAAABBBBBBBBBBBBBBBBBCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCDDDDDDDDDDDEEEFFFGHIJKS X KHGFFEEEDDDDDDDCCCCCCCCBBBBBBBBBBBBBBB
AAAAAAAAABBBBBBCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCDDDDDDDDDDDEEEFFFGQPUVOTY ZQL[MHFEEEEEEEDDDDDDDCCCCCCCCCCCCBBBBBBBBBBBBBB
AAAAAAAAABBBBBBCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCDDDDDDDDDDDEEEFFFGGHLJZ UKHGFEEEEEEEDDDDDDDCCCCCCCCCCCCBBBBBBBBBBBBBB
AAAAAAAAABBBBBBCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCDDDDDDDDDDDEEEFFFGGGGHIKP KHHGGFFEEEEEEEDDDDDDDCCCCCCCCCCCCBBBBBBBBBBBBBB
AAAAAAAAABBBBBBCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCDDDDDDDDDDDEEEFFFGGHIHHHHHIIJKMR VMKJIHHGFFFFFGSGEDDDDDCCCCCCCCCCCCBBBBBBBBBB
AAAAAABBBCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCDDDDDDDDDDDEEEFFFGHK MKJJO N R X YUSR PLV LHHGGHIOJGFEDDDCCCCCCCCCCCCBBBBBBBBB
AAAAABBBCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCDDDDDDDDDDDEEEFFFGH O TN S NKJKR LLQMNHEEDDDCCCCCCCCCCCCBBBBBBBBB
AAAAABBBCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCDDDDDDDEEEFFFGGHHIN Q UMWGEEDDDCCCCCCCCCCCCBBBBBBB
AAAABBBCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCDDDDDEEEFFFGGHIJKLOT [JGFFEEEDDDCCCCCCCCCCCCBBBBBB
AAAABCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCDDDDDEEEFFFGGHHYV RQU QMJHGGFEEEDDDCCCCCCCCCCCCBBBBB
AAABCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCDDDDDEEFFJHFFFFFFFHHGGGGGHIJN JHHGFEEDDDDDCCCCCCCCCCCCBBBBB
AAABCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCDDDDDEEFFHKKHGGGHHMJHGGGGGHHHKKRR UQ L HFEDDDDDCCCCCCCCCCCCBBBB
AABCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCDDDDDEEEFFHKKMRKNIJLVS JJKIIIIJLR YNHFEEDDDDDCCCCCCCCCCCCBBBB
AABCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCDDDDDEEEFFFGHIJKOU O O PR LLJJJKL OIHFFEDDDDDCCCCCCCCCCCCBBBB
AACCCDDDDDDDDDDDDDEEEFFFGGHIJMR RMLMN NTFEEDDDDDCCCCCCCCCCCCBBBB
AACCCDDDDDDDDDDDDDEEEFFFGGHHKONSZ QPR NJGFEEDDDDDCCCCCCCCCCCCCCCC
ABCDDDDDDDDDDDEEEFFFGGHIJMKQ VX HFFEDDDDDDDCCCCCCCCCCCCCCCC
ACDDDDDDDDDEFFFGGHHIKZOPPS HGFEEDDDDDDDCCCCCCCCCCCCCCCC
ADEEEFFFGHIGGGGGHHHIIJLNY TJHGFEEEDDDDDDDCCCCCCCCCCCCCCCC
A PLJHGGFEEEDDDDDDDCCCCCCCCCCCCCCCC
ADEEEFFFGHIGGGGGHHHIIJLNY TJHGFEEEDDDDDDDCCCCCCCCCCCCCCCC
ACDDDDDDDDDEFFFGGHHIKZOPPS HGFEEDDDDDDDCCCCCCCCCCCCCCCC
ABCDDDDDDDDDEEEFFFGGHIJMKQ VX HFFEDDDDDDDCCCCCCCCCCCCCCCC
AACCCDDDDDDDDDDDEEEFFFGGHHKONSZ QPR NJGFEEDDDDDDDCCCCCCCCCCCCCCCC
AACCCDDDDDDDDDDDDDEEEFFFGGHIJMR RMLMN NTFEEDDDDDDDCCCCCCCCCCCCCCCC
AABCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCDDDDDEEEFFFGHIJKOU O O PR LLJJJKL OIHFFEDDDDDDDCCCCCCCCCCCCCCCC
AABCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCDDDDDEEFFHKKMRKNIJLVS JJKIIIIJLR YNHFEEDDDDDDDCCCCCCCCCCCCCCCC
AAABCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCDDDDDEEFFHKKHGGGHHMJHGGGGGHHHKKRR UQ L HFEDDDDDCCCCCCCCCCCCCCCC
AAABCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCDDDDDEEFFJHFFFFFFFHHGGGGGHIJN JHHGFEEDDDDDCCCCCCCCCCCCCCCC
AAAABCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCDDDDDEEEFFFGGHHYV RQU QMJHGGFEEEDDDCCCCCCCCCCCCCCCC
AAAABBBCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCDDDDDEEEFFFGGHIJKLOT [JGFFEEEDDDDDCCCCCCCCCCCCBBBBBB
AAAABBBCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCDDDDDEEEFFFGGHHIN Q UMWGEEDDDDDCCCCCCCCCCCCBBBBBB
AAAABBBCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCDDDDDDDEEEFFFGH O TN S NKJKR LLQMNHEEDDDCCCCCCCCCCCCBBBBBBB
AAAABBBCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCDDDDDDDEEEFFFGHK MKJJO N R X YUSR PLV LHHGGHIOJGFEDDDCCCCCCCCCCCCBBBBBBB
AAAAAABBBBCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCDDDDDDDDDDDEEEFFFGHIHHHHHIIJKMR VMKJIHHGFFFFFGSGEDDDDDCCCCCCCCCCCCBBBBBBBBB
```

Unit Tests Optimizer

Um die Funktion der Optimizer zu garantieren, haben wir Unit tests geschrieben.

Die Unit Tests sind alle bestanden. Den Code der Unit Tests kann auf dem VSC eingesehen werden.

```
running 6 tests
test core::ir::optimizers::test::optimize_clear::test_loop_vec_expression_decptr_1_test_loop_vec_expression_decptr_1_expects ... ok
test core::ir::optimizers::test::copy_optimizer::vec_expression_loop_vec_expression_decval_1_expression_incptr_1_expression_decptr_1_vec_expression_mulval_1_1_expression_clear_expects ... ok
test core::ir::optimizers::test::optimize_clear::test_loop_vec_expression_decptr_1_expression_incptr_1_test_loop_vec_expression_decptr_1_expression_incptr_1_expects ... ok
test core::ir::optimizers::test::optimize_clear::test_loop_vec_expression_decval_1_test_expr_expression_clear_expects ... ok
test core::ir::optimizers::test::optimize_clear::test_loop_vec_expression_incptr_1_test_loop_vec_expression_incptr_1_expects ... ok

test result: ok. 6 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out; finished in 0.00s

Running unittests bin/transpilers/c.rs (target\debug\deps\c-ed30cb07c3e3d7f.exe)

running 0 tests

test result: ok. 0 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out; finished in 0.00s

Running unittests bin/transpilers/rust.rs (target\debug\deps\rust-5376a2bbe0dde1e.exe)

running 0 tests

test result: ok. 0 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out; finished in 0.00s

Running unittests bin/vm.rs (target\debug\deps\vm-11493ac4f2078c9e.exe)

running 0 tests

test result: ok. 0 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out; finished in 0.00s

Doc-tests bf

running 0 tests

test result: ok. 0 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out; finished in 0.00s
```

Abschlussbericht

Erreichte Ziele

Wir freuen uns, mitteilen zu können, dass wir alle unsere Ziele in diesem Projekt erreicht haben. Die Implementierung des Brainfuck Compilers und die Visualisierung auf der Embedded Hardware Tinkerforge wurden erfolgreich abgeschlossen. Wir haben viel über Java, Brainfuck und systematische Abläufe beim Erstellen von Codes gelernt und sind an den Herausforderungen gewachsen.

Schwierigkeiten

Einige Schwierigkeiten im Zeitmanagement bei der Programmierung in Java führten jedoch zu Verzögerungen in der Dokumentation des Projekts. Wir sind uns bewusst, dass wir uns besser und genauer auf den Projektauftrag konzentrieren müssen, um solche Schwierigkeiten zu vermeiden. Wir haben uns auch vorgenommen, das Projekt von Anfang an sorgfältiger zu dokumentieren, um besser auf dem Laufenden zu bleiben und eine effektive Projektdokumentation zu erstellen.

Verbesserungsvorschläge

Für zukünftige Projekte empfehlen wir eine sorgfältige Einschätzung des Projekts und eine klare Definition der Ziele, um sicherzustellen, dass das Projekt reibungslos verläuft. Wir schlagen auch vor, regelmässige Meetings und Überprüfungen des Fortschritts durchzuführen, um auf dem Laufenden zu bleiben und Verzögerungen zu minimieren.

Schlussfolgerung

Zusammenfassend war dies ein erfolgreiches Projekt, das uns viel Wissen und Erfahrung gebracht hat. Wir sind stolz darauf, alle unsere Ziele erreicht zu haben und danken allen, die uns dabei unterstützt haben.

Schlussenteil

Ausblick

Das erfolgreiche Abschliessen dieses Projekts gibt uns die Möglichkeit, uns auf zukünftige Projekte zu freuen und zu planen. Wir haben in diesem Projekt viel gelernt und sind sicher, dass dieses Wissen uns bei zukünftigen Herausforderungen helfen wird. Wir planen, das Wissen und die Erfahrung, die wir in diesem Projekt gewonnen haben, in unseren zukünftigen Projekten anzuwenden und weiterzuentwickeln.

Wir werden auch weiterhin daran arbeiten, unsere Fähigkeiten zu verbessern und uns auf unsere individuellen Interessen und Stärken zu konzentrieren. Wir sind dankbar für die Chance, an diesem Projekt teilgenommen zu haben und freuen uns darauf, unser Wissen in zukünftigen Projekten anzuwenden. Besonders in der Programmiersprache Java erwarten wir weitere Projekte und sind mit diesen Erfahrungen im gepäck nun besser vorbereitet.

Ein Ausblick auf dieses Projekt spezifisch: wir können die Fehlermeldung von der Rust Runtime vorbeugen. Weiter sehen wir grosses Potential in der vereinfachung und Automatisierung der Pipeline:

- Input zur Pipeline autimatisieren

- File Outputstream vordefinieren
- Ganzer Pipeline Ablauf mit so wenig Interaktion wie möglich gestalten

Schlusswort

Wir möchten uns bei allen bedanken, die uns bei diesem Projekt unterstützt haben. Ein besonderer Dank geht an unser Team, das engagiert zusammengearbeitet hat, um dieses Projekt erfolgreich abzuschliessen. Wir möchten auch unseren Betreuern und Lehrern danken, die uns während des Projekts unterstützt und inspiriert haben.

Dieses Projekt hat uns nicht nur gezeigt, wie wichtig eine sorgfältige Planung und Durchführung von Projekten ist, sondern auch, wie wichtig es ist, als Team zusammenzuarbeiten und sich gegenseitig zu unterstützen. Wir haben viel gelernt und sind stolz auf das, was wir erreicht haben.

Abschliessend möchten wir sagen, dass wir uns auf zukünftige Projekte freuen und uns darauf konzentrieren werden, unser Wissen und unsere Fähigkeiten zu erweitern und zu verbessern. Wir sind dankbar für die Erfahrungen, die wir in diesem Projekt gemacht haben und freuen uns auf die Herausforderungen, die uns in Zukunft erwarten.

Anhang

Orientation Dokumentation Tinkerforge HAT: <https://www.tinkerforge.com/en/>

TestFiles

[bfeer](#) [bfeerInput](#) [Mandelbrot](#) [MandelbrotInput](#)